Computer Supported Modeling and Reasoning

David Basin, Achim D. Brucker, Jan-Georg Smaus, and Burkhart Wolff

April 2005

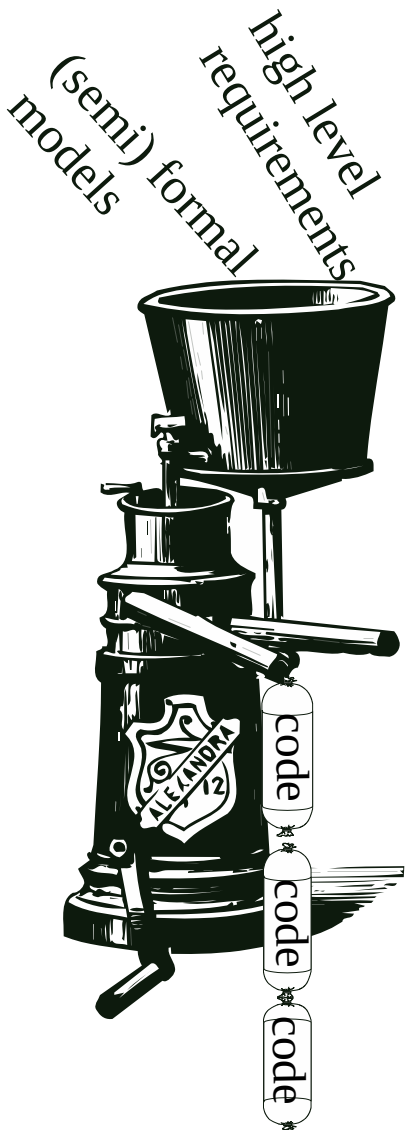
http://www.infsec.ethz.ch/education/permanent/csmr/

Introduction

David Basin and Burkhart Wolff

What this Course is about

- Modeling in Mathematics and Computer Science
  - logics and proofs:
    input: logics, deductive systems
    output: (meta)-theorems
  - program programs and verifications:
    input: specifications, programs, properties
    output: correctness guarantees
- Technically: mechanization and application
- Generally: making logic come to life and useful as a general tool.



Why this Course Matters

Academic motivation: deepen knowledge of logic and formal reasoning. Gain insights into philosophical foundations such as Reductionism and Hilbert's Program

Why this Course Matters

Academic motivation: deepen knowledge of logic and formal reasoning. Gain insights into philosophical foundations such as Reductionism and Hilbert's Program

Practical motivation: verification and formal methods

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

Why this Course Matters (2)

In general:

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

Relationship to Other Courses

Logic: deduction, foundations, and applications

Software engineering: specification, refinement, verification

Hardware: formalizing and reasoning about circuit models

Artificial Intelligence: knowledge representation, reasoning, deduction

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

Overview: Five Parts

1. Logics (propositional, first-order, higher-order)

Overview: Five Parts

1. Logics (propositional, first-order, higher-order)
2. The "Metalogical Approach":
   Representing logics in logics

Overview: Five Parts

1. Logics (propositional, first-order, higher-order)
2. The "Metalogical Approach":
   Representing logics in logics
3. Theorem Proving with Isabelle:
   proof strategies, machine supported theory development

Overview: Five Parts

1. Logics (propositional, first-order, higher-order)
2. The "Metalogical Approach":
   Representing logics in logics
3. Theorem Proving with Isabelle:
   proof strategies, machine supported theory development
4. The "Semantic Approach":
   Representing syntax and semantics of logics in logics

Overview: Five Parts

1. Logics (propositional, first-order, higher-order)
2. The "Metalogical Approach":
   Representing logics in logics
3. Theorem Proving with Isabelle:
   proof strategies, machine supported theory development
4. The "Semantic Approach":
   Representing syntax and semantics of logics in logics
5. Applications:
   Case studies in formalizing theories of computer science.

Our presentation roughly follows this conceptual structure.

Overview: Part 1: Logics

1.1 This Introduction

1.2 Propositional Logic (PL)

1.3 Natural Deduction in PL

1.4 First-Order Logic (FOL)

1.5 Natural Deduction in FOL

1.6 Theories in FOL

1.7 Naive Set-Theory in FOL

Overview: Part 2: Metalogical Approach

2.1 Foundation: λ-calculi

2.2 Encoding Syntax in LF

2.3 Encoding Deduction in LF

Overview: Part 3: Theorem Proving with Isabelle

3.1 Basic Deduction: Resolution

3.2 Automated Deduction: Classical Reasoner

3.3 Automated Deduction: Term Rewriting

3.4 The Isabelle Metalogic

3.5 Proof Pragmatics

Overview: Part 4: Semantic Approach

4.1 Foundations of Higher-order Logic (HOL)

4.2 Derived Rules of HOL

4.3 Conservative Extensions in HOL

4.4 Basic Library of HOL

4.5 Fixpoints and Inductive Sets

4.6 Wellfounded Orderings and Recursion

4.7 Arithmetic in HOL

4.8 Datatypes in HOL

Overview: Part 5: Applications

5.1 Encoding Imperative Languages

5.2 Encoding Functional Languages

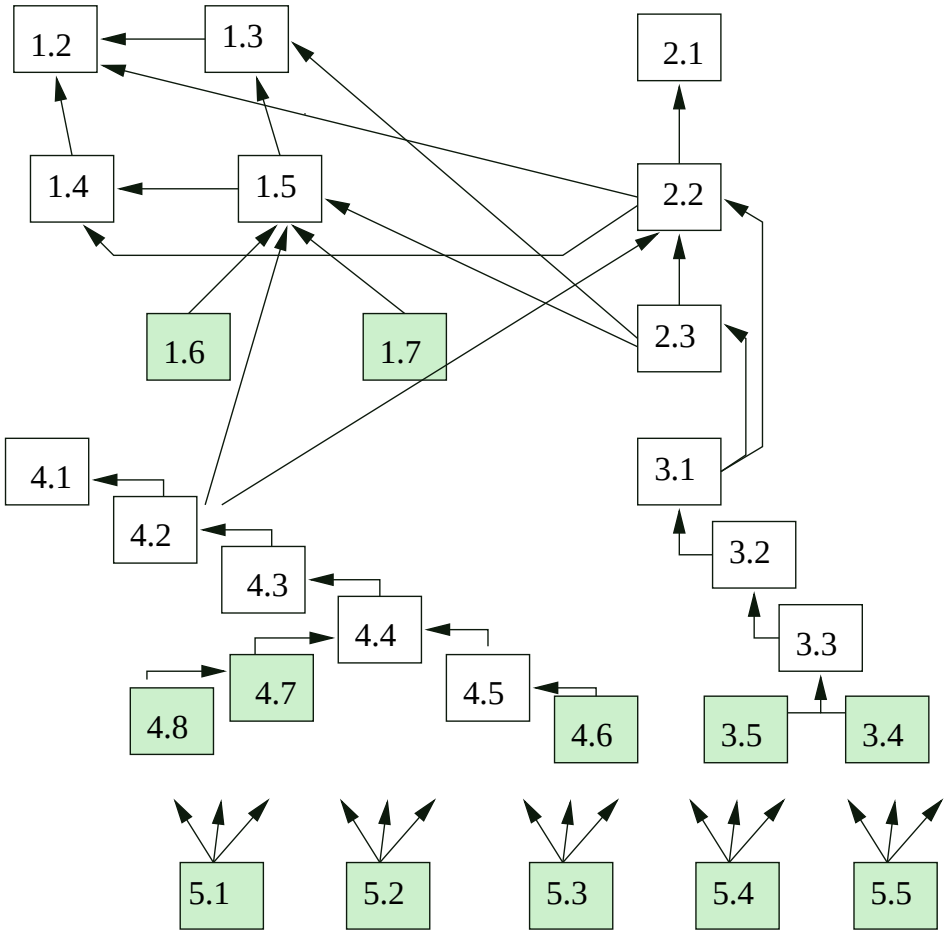
5.3 Refinement

5.4 HOL-OCL

5.5 Other Applications

Overview: Part 6

- Dependencies of the chapters:
- (Green boxes may be skipped at first reading)



Requirements

Some knowledge of logic is a useful prerequisite for this course.

Requirements

Some knowledge of logic is a useful prerequisite for this course.

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

Requirements (2)

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

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

Requirements (2)

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

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

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

More Detailed Explanations

What is Verification?

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

What is Reductionism?

The philosophical position of reductionism, carried to the extreme, says that anything in the world can be reduced to physics and mathematical modeling, which in itself can be reduced to a small number of axioms, and which can be finally reduced to one formula.

While we do not claim such a strong position ourselves, part of the material we present here (in particular HOL and the development of its library) roughly follow the Russels and Whiteheads Principia Mathematica which was deeply influenced by reductionistic ideas.

What is Hilbert's Program?

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

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

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

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

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

For more details:

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

Mechanizing Logic

Meanwhile, a number of theorem proving systems mechanizing logical deduction is available (c.f. The Sixteen Provers of the World; compiled by Freek Wiedijk). The number of Formal Methods systems (i.e. (automated) theorem provers geared towards system and program verification) is even larger.

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

Isabelle has been used for very substantial proofs in pure mathematics (such as the Prime Number Theorem) or computer science (such as the Java Virtual Machine).

What is (a) Logic?

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

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

In a narrower sense, a logic is just a precisely defined language allowing to write down statements (i.e. some of the syntactic entities of this language), together with a predefined inference or deduction mechanism allowing for deducing new statements from established ones. The deduction mechanism is usually represented by logical rules. In this course, we consider Propositional logic, first-order logic, and higher-order logic are three different logics, but also applied logics (called formal specification languages) such as Z or Hoare Logic.

What is a Metalogic?

A metalogic is a logic that is used to formalize syntax, deduction system, semantics and possibly meta-properties (such as correctness, completeness, adequacy, etc) of another logic — called the object logic.

What is a Semantics?

A semantics of a formal language (i.e. a logic, a programming or specification language) is a function that assigns to each element of its syntax a denotation or value.

Given a semantics, the question can be settled when a rule — allowing derivations or deductions or inferences of syntactic elements such as logical statements from other syntactic elements — is correct, namely that it never changes the value of a derived statement.

With the semantic approach we refer to the method consiting in defining an (explicit or implicit) semantic function, usually in terms of a set theory or constructs of similar expressive power, and to derive the logical rules of the object logic from this definition.

What is a Formal Language?

A formal language must have a syntax, i.e. a formally defined set of sequences of elementary symbols and a semantics that give it a formally defined meaning.

The term "formal language" ma be referred to logics, programming or specification languages.

What is a Theory?

A theory is a collection of logical statements in a logic and the set of all logical statements that can be derived from them via the inference of the logic.

Quite often, we will implicitly identify the former (usually finite) set with the (usually infinite) latter one.

A theory is used to model a tiny portion of the "world", let it be gravitation and quantums in physics, or prime numbers in mathematics, or the Java Virtual Machine in computer science.

More information later.

What we Neglect

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

- Gödels completeness and . . .
- . . . incompleteness theorems.

As an introduction we recommend [vD80] or [And86].

Equivalence Classes

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

Propositional Logic

David Basin

Propositional Logic: Overview

- System for formalizing certain valid patterns of reasoning
- Expressions built by combining "atomic propositions" using not, if . . . then . . . , and, or, etc.
- Validity means no counterexample. 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.

Propositional Logic: Overview

- System for formalizing certain valid patterns of reasoning
- Expressions built by combining "atomic propositions" using not, if . . . then . . . , and, or, etc.
- Validity means no counterexample. Depends on form of the expressions ⇒ can make patterns explicit by replacing words by symbols

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

Propositional Logic: Overview

- System for formalizing certain valid patterns of reasoning
- Expressions built by combining "atomic propositions" using not, if . . . then . . . , and, or, etc.
- Validity means no counterexample. Depends on form of the expressions ⇒ can make patterns explicit by replacing words by symbols

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

- What about

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

More Examples

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

More Examples (Which are Valid?)

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

History

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

Formal Systems

Formalization allows us to "turn the crank".

Formal Systems

Formalization allows us to "turn the crank".

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

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

Formal Systems

Formalization allows us to "turn the crank".

Phrases like "from . . . it follows" or "therefore" are formalized 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 in the style of natural deduction.

Formalizing Propositional Logic

- We must formalize
  (a) Language

Formalizing Propositional Logic

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

Formalizing Propositional Logic

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

Formalizing Propositional Logic

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

Propositional Logic: Language

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

Propositional Logic: Language (2)

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

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

The elements of L_P are called (propositional) formulas.

We omit unnecessary brackets.

More Detailed Explanations

What is Validity (of a Pattern of Reasoning)?

Let A and B are symbols (variables) standing for arbitrary propositions. Then

From if A then B and A it follows that B

is valid because it is true regardless of what A and B are. In other words, there is no pair of propositions for A and B that makes this rule false (this would be a counter example). The validity of this (propositional) pattern of reasoning is based on the fact that any proposition must be either true or false; therefore, we have only to check for all combinations in order to establish validity.

A formalization of a "pattern of reasoning" is a logical rule. Therefore, we also speak of a valid rule.

An Invalid Pattern

From if A then B and B it follows that A

is invalid because there is a counterexample:

Let A be "Kim is a man" and B be "Kim is a person".

More Examples

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

More Examples (Which are Valid?)

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

Turning the Crank

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

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

What does Formalization Mean?

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

We study formalization in more detail later.

Proof Systems

By a proof system or deductive system we mean a mechanism that allows for the construction logical statements (e.g. valid formulae) from other statements by purely syntactic means.

In particular, a deductive system can be given by a collection of rules, or a "calculus".

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

We will see for example natural deduction calculi and sequent calculi for various logics in this course.

Soundness and Completeness

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

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

What is a Meta-theorem?

A metatheorem is a theorem about a proof system, as opposed to a theorem derived within the proof system. The statements "proof system XYZ is sound" or "proof system XYZ is complete" are meta-theorems.

What is a Language?

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

What is Semantics?

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

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

We go into more detail later.

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

What does Open Assumption Mean?

For example, all logical statements at the leaves of the proof:

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

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

What is $\vdash$?

This symbol is used to mark a form of logical statement. By writing $A \to (B \to C), A, B \vdash C$, we assert that C can be derived in this proof system under the open assumptions $A \to (B \to C)$, A, B.

This form of logical statements gives rise to explained later.

Why is this Example Abstract?

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

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

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

How to Read these Rules

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

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

How to Read these Rules (2)

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

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

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

Making Assumptions

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

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

It is thus unproblematic to "make" assumptions.

Propositional Variables

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

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

What is a Formula?

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

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

Associativity and Precedences

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

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

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

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

Why Smallest Set?

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

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

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

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

Introduction and Elimination

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

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

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

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

Validity Revisited

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

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

This is consistent with the

earlier intuitive explanation of validity of a formula. Details can be found in any textbook on logic [vD80].

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

Schematic Rules

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

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

We will see more about the use of metavariables later.

Can we Prove Anything . . . ?

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

None of the rules involves discharging an assumption.

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

Consequently, the answer is no.

Object vs. Meta

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

So you Find this Strange!

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

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

where both A and B are replaced with P.

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

$A \to B \to A$

The rule(s):

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

The proof:

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

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

The rules:

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

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

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

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

The proof:

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

$(A \to B \to C) \to (A \to B) \to A \to C$

The rules:

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

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

The proof:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{[(A \to B \to C)]^1 \quad [A]^3}{B \to C}\ {\to}\text{-}E \quad \dfrac{[(A \to B)]^2 \quad [A]^3}{B}\ {\to}\text{-}E}{C}\ {\to}\text{-}E}{A \to C}\ {\to}\text{-}I^3}{(A \to B) \to A \to C}\ {\to}\text{-}I^2}{(A \to B \to C) \to (A \to B) \to A \to C}\ {\to}\text{-}I^1$$

Falsity

The symbol $\perp$ stands for "false".

No Introduction Rule for $\perp$

The symbol $\perp$ stands for "false".

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

Connectives

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

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

Negation

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

The Rules for $\neg$

The rule

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

is simply an instance of $\rightarrow$-*E* (since $\neg A$ is shorthand for $A \rightarrow \bot$).

Likewise, the rule

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

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

The Rule $\neg$-E

For negation, it is common to have a rule

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

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

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

Peirce's Law Valid?

Yes, simply check the truth table:

A	B	$((A \rightarrow B) \rightarrow A) \rightarrow A$
$True$	$True$	$True$
$True$	$False$	$True$
$False$	$True$	$True$
$False$	$False$	$True$

Peirce's Law Provable?

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

Intuitionistic versus Classical Logic

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

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

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

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

Axiom of the Excluded Middle

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

Reductio ad absurdum

The rule

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

is called reduction ad absurdum.

The Classical Rule

The rule

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

corresponds to a formulation in Isabelle.

Sequent Notation

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

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

The notation $\Gamma \vdash A$ is called sequent notation. However, this should not be confused with the sequent calculus (we will consider it later). The sequent calculus is based on sequents, which are syntactic entities of the form $A_1, \ldots, A_n \vdash B_1, \ldots, B_m$, 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 call natural deduction using a sequent notation.

Axioms vs. Rules

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

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

Assumptions

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

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

Weakening

The rule *weaken* is

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

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.

Deriving $\wedge$-E

As an example, consider

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

This rule can be derived as follows:

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

Which Rule to Choose?

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

The question will be addressed more systematically later.

Goals to Axioms

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

Working with Assumptions

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

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

Propositional Logic: Natural Deduction

David Basin, Burkhart Wolff, and Jan-Georg Smaus

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 discharging 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.

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.

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

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.

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

A proof is a derivation with no (open) assumptions.

Natural Deduction: an Abstract Example

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

Natural Deduction: an Abstract Example

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

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

How do you read these rules?

Natural Deduction: an Abstract Example

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

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

How about this one?

Natural Deduction: an Abstract Example

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

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

How about this one?

N.B. $\alpha, \beta, \gamma, \delta$ just name the rules.

Proof of ♥

The rules:

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

The proof:

Proof of ♥

The rules:

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

The proof:

$$\diamondsuit$$

We make an assumption. The assumption is now open.

Proof of ♥

The rules:

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

The proof:

$$\frac{\diamondsuit}{\clubsuit}\,\alpha$$

We apply α.

Proof of ♥

The rules:

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

The proof:

$$\frac{\diamondsuit}{\clubsuit}\,\alpha \qquad \frac{\diamondsuit}{\spadesuit}\,\beta$$

Similarly with β.

Proof of ♥

The rules:

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

The proof:

$$\frac{\dfrac{\diamondsuit}{\clubsuit}\,\alpha \qquad \dfrac{\diamondsuit}{\spadesuit}\,\beta}{\heartsuit}\,\gamma$$

We apply γ.

Proof of ♥

The rules:

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

The proof:

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

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

Deductive System: Rules of Propositional Logic

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

Some rules introduce, others eliminate connectives.

Rules of Propositional Logic: Conjunction

- Rules of two kinds: introduce connectives

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

Rules of Propositional Logic: Conjunction

- Rules of two kinds: introduce and eliminate 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 of Propositional Logic: Conjunction

- Rules of two kinds: introduce and eliminate 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.
- Why valid? If all assumptions are true, then so is conclusion

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

Example Derivation with Conjunction

The rules:

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

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

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

Example Derivation with Conjunction

The rules:

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

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

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

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

Example Derivation with Conjunction

The rules:

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

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

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

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

Example Derivation with Conjunction

The rules:

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

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

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

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

Example Derivation with Conjunction

The rules:

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

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

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

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

Example Derivation with Conjunction

The rules:

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

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

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

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

Can we prove anything with just these three rules?

Rules of Propositional Logic: Implication

- Rules

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

Rules of Propositional Logic: Implication

- Rules

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

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

Rules of Propositional Logic: Implication

- Rules

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

- $\to$-E is also called modus ponens.
- $\to$-I formalizes (bottom-up) strategy:
  To derive $A \to B$, derive B under the additional (local) assumption A.
  Top-down: we may discharge 0 or more occurences of A.

A Simple Proof

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

$$P$$

A 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?

Examples with Conjunction and Implication

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

Object versus Meta: variables here can either represent object variables or metavariables.

Disjunction

- Rules

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

Disjunction

- Rules

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

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

Disjunction: Example

- Rules

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

- Example: formalize and prove

  When it rains then I wear my jacket.

  When it snows then I wear my jacket.

  It is raining or snowing.

  Therefore I wear my jacket.

Falsity and Negation

- Falsity

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

No introduction rule!

Falsity and Negation

- Falsity

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

No introduction rule!

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

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

Intuitionistic versus Classical Logic

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

Intuitionistic versus Classical Logic

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

$$A \vee \neg A \quad \text{or} \quad \dfrac{\begin{array}{c}[\neg A] \\ \vdots \\ \bot\end{array}}{A}\,RAA \quad \text{or} \quad \dfrac{\begin{array}{c}[\neg A] \\ \vdots \\ A\end{array}}{A}\,\textit{classical}.$$

Example of Classical Reasoning

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

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

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

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

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

Overview of Rules

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

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

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

Deductive System: Derived Rules

Using the basic rules, we can derive new rules.

Example: Resolution rule.

Deductive System: Derived Rules

Using the basic rules, we can derive new rules.

Example: Resolution rule.

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

It looks like this.

Deductive System: Derived Rules

Using the basic rules, we can derive new rules.

Example: Resolution rule.

$$\neg S$$

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

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

Deductive System: Derived Rules

Using the basic rules, we can derive new rules.

Example: Resolution rule.

$$\neg S$$

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

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!

Deductive System: Derived Rules

Using the basic rules, we can derive new rules.

Example: Resolution rule.

$$\neg S \qquad S$$

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

So now S.

Deductive System: Derived Rules

Using the basic rules, we can derive new rules.

Example: Resolution rule.

$$\frac{\neg S \quad S}{\bot}\ \rightarrow\text{-}E$$

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

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

Deductive System: Derived Rules

Using the basic rules, we can derive new rules.

Example: Resolution rule.

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

$$\frac{R \vee S \quad R \quad \dfrac{\dfrac{\neg S \quad S}{\bot}\,{\to}\text{-}E}{R}\,\bot\text{-}E}{R}\,\vee\text{-}E$$

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

Deductive System: Derived Rules

Using the basic rules, we can derive new rules.

Example: Resolution rule.

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

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

Alternative Deductive System Using Sequent Notation

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

Sequent Rules (for $\rightarrow/\wedge$ Fragment)

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

Rules for assumptions and weakening

Sequent Rules (for $\rightarrow / \wedge$ Fragment)

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

Rules for assumptions and weakening

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

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

More rules can be derived.

Example: Refinement Style with Metavariables

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

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

Example: Refinement Style with Metavariables

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

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

Example: Refinement Style with Metavariables

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

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

Example: Refinement Style with Metavariables

$$\dfrac{\dfrac{\dfrac{A \wedge (B \wedge C) \vdash A \wedge ?X}{A \wedge (B \wedge C) \vdash A}\ \wedge\text{-}EL \qquad \dfrac{}{A \wedge (B \wedge C) \vdash C}\ \wedge\text{-}ER}{A \wedge (B \wedge C) \vdash A \wedge C}\ \wedge\text{-}I}{\vdash A \wedge (B \wedge C) \rightarrow A \wedge C}\ \rightarrow\text{-}I$$

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

Example: Refinement Style with Metavariables

$$\dfrac{\dfrac{\dfrac{A \wedge (B \wedge C) \vdash A \wedge ?X}{A \wedge (B \wedge C) \vdash A}\ \wedge\text{-}EL \qquad \dfrac{\dfrac{}{A \wedge (B \wedge C) \vdash (?Y \wedge C)}\ \wedge\text{-}ER}{A \wedge (B \wedge C) \vdash C}\ \wedge\text{-}ER}{A \wedge (B \wedge C) \vdash A \wedge C}\ \wedge\text{-}I}{\vdash A \wedge (B \wedge C) \rightarrow A \wedge C}\ \rightarrow\text{-}I$$

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

Example: Refinement Style with Metavariables

$$
\dfrac{\dfrac{\dfrac{A \wedge (B \wedge C) \vdash A \wedge ?X}{A \wedge (B \wedge C) \vdash A}\ \wedge\text{-}EL \qquad \dfrac{\dfrac{A \wedge (B \wedge C) \vdash ?Z \wedge (?Y \wedge C)}{A \wedge (B \wedge C) \vdash (?Y \wedge C)}\ \wedge\text{-}ER}{A \wedge (B \wedge C) \vdash C}\ \wedge\text{-}ER}{A \wedge (B \wedge C) \vdash A \wedge C}\ \wedge\text{-}I}{\vdash A \wedge (B \wedge C) \rightarrow A \wedge C}\ \rightarrow\text{-}I
$$

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

Example: Refinement Style with Metavariables

$$\dfrac{\dfrac{\dfrac{A \wedge (B \wedge C) \vdash A \wedge ?X}{A \wedge (B \wedge C) \vdash A}\ \wedge\text{-}EL \qquad \dfrac{\dfrac{A \wedge (B \wedge C) \vdash ?Z \wedge (?Y \wedge C)}{A \wedge (B \wedge C) \vdash (?Y \wedge C)}\ \wedge\text{-}ER}{A \wedge (B \wedge C) \vdash C}\ \wedge\text{-}ER}{A \wedge (B \wedge C) \vdash A \wedge C}\ \wedge\text{-}I}{\vdash A \wedge (B \wedge C) \rightarrow A \wedge C}\ \rightarrow\text{-}I$$

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

Comments about Proof Refinement

This crazy way of carrying out proofs is the (standard) way, which is used in many proof assistants (as Isabelle)!

- Refinement style is also called backward style proofs
- Refinement style means we work from goals to axioms
- metavariables are used to delay substitions

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

How Are ND Proofs Built?

ND proofs build derivations under (possibly temporary) assumptions.

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:

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

Alternative Formalization Using Sequents

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

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

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

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

Two representations equivalent. Sequent notation seems simpler in practice.

Example: Refinement Style with Metavariables

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

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

We went through this example in detail last lecture.

Comments about Refinement

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

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

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

More Detailed Explanations

What are ND Systems and Proofs?

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

What is Sequent Notation?

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

Sequent Notation and Isabelle

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

First-Order Logic

David Basin, Burkhart Wolff, and Jan-Georg Smaus

First-Order Logic: Overview

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

First-Order Logic: Overview

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

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

First-Order Logic: Overview

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

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

Cannot talk about relations and functions.

Cannot say things like "the state alternates over time".

First-Order Logic: Overview

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

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

Cannot talk about relations and functions.

Cannot say things like "the state alternates over time".

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

Variables: Intuition

In first-order logic, we talk about "elements of a universe of discourse" and their "properties".

A variable in first-order logic stands for a element of the universe.

Variables: Intuition

In first-order logic, we talk about "elements of a universe of discourse" and their "properties".

A variable in first-order logic stands for a element of the universe.

This is in contrast to propositional logic.

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

Predicates: Intuition

A predicate denotes a property/relation.

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

Predicates: Intuition

A predicate denotes a property/relation.

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

Propositional connectives are used to build statements

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

Predicates: Intuition

A predicate denotes a property/relation.

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

Propositional connectives are used to build statements

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

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

Predicates: Intuition

A predicate denotes a property/relation.

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

Propositional connectives are used to build statements

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

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

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

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 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 likes y but not vice versa

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

Predicates: Intuition (2)

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

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

Functions: Intuition

- A constant stands for a "fixed thing" in a universe.

Functions: Intuition

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

Function	arity	expresses . . .
0		
s		
$+$		

Functions: Intuition

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

Function	arity	expresses . . .
0	nullary	
s	unary	
$+$	binary	

Functions: Intuition

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

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

Quantifiers: Intuition

- A variable stands for "some element" in the universe of discourse. Quantifiers $\forall, \exists$ are used to speak about all or some members of this universe.

Quantifiers: Intuition

- A variable stands for "some element" in the universe of discourse. Quantifiers $\forall, \exists$ are used to speak about all or some members of this universe.
- Examples: Are they satisfiable? valid?

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

Quantifiers: Intuition

- A variable stands for "some element" in the universe of discourse. Quantifiers $\forall, \exists$ are used to speak about all or some members of this universe.
- Examples: Are they satisfiable? valid?

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

Quantifiers: Intuition

- A variable stands for "some element" in the universe of discourse. Quantifiers $\forall, \exists$ are used to speak about all or some members of this universe.
- Examples: Are they satisfiable? valid?

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

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

Quantifiers: Intuition

- A variable stands for "some element" in the universe of discourse. Quantifiers $\forall, \exists$ are used to speak about all or some members of this universe.
- Examples: Are they satisfiable? valid?

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

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

Quantifiers: Intuition

- A variable stands for "some element" in the universe of discourse. Quantifiers $\forall, \exists$ are used to speak about all or some members of this universe.
- Examples: Are they satisfiable? valid?

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

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

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

Quantifiers: Intuition

- A variable stands for "some element" in the universe of discourse. Quantifiers $\forall, \exists$ are used to speak about all or some members of this universe.
- Examples: Are they satisfiable? valid?

$\forall x.\, \exists y.\, y * 2 = x$	true for rationals
$x < y \rightarrow \exists z.\, x < z \wedge z < y$	true for any dense order
$\exists x.\, x \neq 0$	true for universes with more than one element

Quantifiers: Intuition

- A variable stands for "some element" in the universe of discourse. Quantifiers $\forall, \exists$ are used to speak about all or some members of this universe.
- Examples: Are they satisfiable? valid?

$\forall x.\, \exists y.\, y * 2 = x$	true for rationals
$x < y \rightarrow \exists z.\, x < z \wedge z < y$	true for any dense order
$\exists x.\, x \neq 0$	true for universes with more than one element

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

Quantifiers: Intuition

- A variable stands for "some element" in the universe of discourse. Quantifiers $\forall, \exists$ are used to speak about all or some members of this universe.
- Examples: Are they satisfiable? valid?

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

First-Order Logic: Syntax

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

First-Order Logic: Syntax

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

First-Order Logic: Syntax

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

Terms in First-Order Logic

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

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

Formulae in First-Order Logic

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

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

The formulae 2 above are called atoms.

Variable Occurrences

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

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

- Example:

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

Which are bound?

Variable Occurrences

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

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

- Example:

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

Which are bound?

Which are free?

Variable Occurrences

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

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

- Example:

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

Which are bound?

Which are free?

Which are binding?

Variable Occurrences

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

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

- Example:

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

Which are bound?

Which are free?

Which are binding?

First-Order Logic: Semantics

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

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

The Value of Terms

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

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

The Value of Formulae

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

$$
\begin{aligned}
\mathcal{A}(\bot) &= 0 \\
\mathcal{A}(p(t_1, \ldots, t_n)) &= \begin{cases} 1 & \text{if } (\mathcal{A}(t_1), \ldots \mathcal{A}(t_n)) \in p^{\mathcal{A}} \\ 0 & \text{otherwise} \end{cases} \\
\mathcal{A}(\neg\phi) &= \begin{cases} 1 & \text{if } \mathcal{A}(\phi) = 0 \\ 0 & \text{otherwise} \end{cases} \\
&\vdots
\end{aligned}
$$

The Value of Formulae (2)

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

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

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

Models

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

Models

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

Models

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

Models

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

Models

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

There are alternative ways to formulate this.

An Example

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

An Example

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

A model:

$$\begin{aligned} U_{\mathcal{A}} &= \mathcal{N} \\ p^{\mathcal{A}} &= \{(m,n) \mid m < n\} \\ s^{\mathcal{A}}(x) &= x + 1 \end{aligned}$$

An Example

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

A model:

$$\begin{aligned} U_{\mathcal{A}} &= \mathcal{N} \\ p^{\mathcal{A}} &= \{(m,n) \mid m < n\} \\ s^{\mathcal{A}}(x) &= x + 1 \end{aligned}$$

Not a model:

$$\begin{aligned} U_{\mathcal{A}} &= \{\mathtt{a}, \mathtt{b}, \mathtt{c}\} \\ p^{\mathcal{A}} &= \{(\mathtt{a}, \mathtt{b}), (\mathtt{a}, \mathtt{c})\} \\ s^{\mathcal{A}} &= \text{“the identity function”} \end{aligned}$$

Towards a Deductive System

Consider an "ordinary" mathematical proof of

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

Towards a Deductive System

Consider an "ordinary" mathematical proof of

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

In natural language, quantifiers are often implicit.

Proof:

Towards a Deductive System

Consider an "ordinary" mathematical proof of

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

In natural language, quantifiers are often implicit.

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

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

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

Towards a Deductive System

Consider an "ordinary" mathematical proof of

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

In natural language, quantifiers are often implicit.

Proof: Consider an arbitrary x ($\forall$-*I*) where $x > 2$ ($\rightarrow$-*I*).
Then $x = 2 + y$ for some $y > 0$ and hence

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

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

Towards a Deductive System

Consider an "ordinary" mathematical proof of

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

In natural language, quantifiers are often implicit.

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

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

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

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

Weaker Statement

Even easier to prove the weaker statement
$\exists x.\, x > 2 \rightarrow x^2 > 4$.

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

Weaker Statement

Even easier to prove the weaker statement
$\exists x.\, x > 2 \rightarrow x^2 > 4$.

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

Intuition: existential statements are proven by giving a witness.

First-Order Logic: Natural Deduction

David Basin, Burkhart Wolff, and Jan-Georg Smaus

First-Order Logic: Deductive System

First-order logic is a generalization of propositional logic. All the rules of propositional logic carry over.

But we must introduce rules for the quantifiers.

Universal Quantification ($\forall$): 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 $*$ means: x must be arbitrary.

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 $*$ means: x must be arbitrary.

Note that rules are schematic.

Universal Quantification: Side Condition

What does arbitrary mean? Consider the following "proof"

$$x = 0$$

Universal Quantification: Side Condition

What does arbitrary mean? Consider the following "proof"

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

Universal Quantification: Side Condition

What does arbitrary mean? Consider the following "proof"

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

Universal Quantification: Side Condition

What does arbitrary mean? Consider the following "proof"

$$\dfrac{\dfrac{\dfrac{[x = 0]^1}{\forall x.\, x = 0}\ \forall\text{-}I}{x = 0 \rightarrow \forall x.\, x = 0}\ \rightarrow\text{-}I^1}{\forall x.\, (x = 0 \rightarrow \forall x.\, x = 0)}\ \forall\text{-}I$$

Universal Quantification: Side Condition

What does arbitrary mean? Consider the following "proof"

$$\dfrac{\dfrac{\dfrac{\dfrac{[x = 0]^1}{\forall x.\, x = 0}\ \forall\text{-}I}{x = 0 \rightarrow \forall x.\, x = 0}\ \rightarrow\text{-}I^1}{\forall x.\, (x = 0 \rightarrow \forall x.\, x = 0)}\ \forall\text{-}I}{0 = 0 \rightarrow \forall x.\, x = 0}\ \forall\text{-}E$$

Universal Quantification: Side Condition

What does arbitrary mean? Consider the following "proof"

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

Universal Quantification: Side Condition

What does arbitrary mean? Consider the following "proof"

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

Universal Quantification: Side Condition

What does arbitrary mean? Consider the following "proof"

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

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

Another Proof? (1)

Is the following a proof? Is the conclusion valid?

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

Another Proof? (1)

Is the following a proof? Is the conclusion valid?

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

Conclusion is not valid.

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

Another Proof? (1)

Is the following a proof? Is the conclusion valid?

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

Proof is incorrect.

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

Another Proof? (2)

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

Another Proof? (2)

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

Another Proof? (2)

$$\dfrac{\dfrac{\forall x.\, A(x) \wedge B(x)}{A(x) \wedge B(x)}\ \forall\text{-}E}{A(x)}\ \wedge\text{-}EL$$

Another Proof? (2)

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

Another Proof? (2)

$$\dfrac{\dfrac{\dfrac{\forall x.\, A(x) \land B(x)}{A(x) \land B(x)}\ \forall\text{-}E}{A(x)}\ \land\text{-}EL}{\forall x.\, A(x)}\ \forall\text{-}I \qquad \dfrac{\dfrac{\dfrac{\forall x.\, A(x) \land B(x)}{A(x) \land B(x)}\ \forall\text{-}E}{B(x)}\ \land\text{-}ER}{\forall x.\, B(x)}\ \forall\text{-}I$$

Another Proof? (2)

$$
\dfrac{
\dfrac{
\dfrac{
\dfrac{\forall x.\, A(x) \land B(x)}{A(x) \land B(x)}\,\forall\text{-}E
}{A(x)}\,\land\text{-}EL
}{\forall x.\, A(x)}\,\forall\text{-}I
\qquad
\dfrac{
\dfrac{
\dfrac{\forall x.\, A(x) \land B(x)}{A(x) \land B(x)}\,\forall\text{-}E
}{B(x)}\,\land\text{-}ER
}{\forall x.\, B(x)}\,\forall\text{-}I
}{(\forall x.\, A(x)) \land (\forall x.\, B(x))}\,\land\text{-}I
$$

Another Proof? (2)

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

Another Proof? (2)

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

Yes (check side conditions of $\forall$-*I*).

Aside: $A \leftrightarrow B$

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

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

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

You could do this as an exercise!

Proof?

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

Proof?

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

Yes, but only if x not free in A.

Proof?

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

Yes, but only if x not free in A.

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

Existential Quantification

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

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

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

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

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

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

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

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

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

$\exists$-*I* as a Derived Rule

The rule:

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

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

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

$\exists$-*I* as a Derived Rule

The rule:

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

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

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

$\exists$-*I* as a Derived Rule

The rule:

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

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

$$\bot$$

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

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

$\exists$-*I* as a Derived Rule

The rule:

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

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

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

We apply $\forall$-*E*.

$\exists$-*I* as a Derived Rule

The rule:

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

$$\dfrac{\dfrac{\dfrac{\forall x.\, \neg A(x)}{\neg A(t)}\ \forall\text{-}E \qquad A(t)}{\bot}\ \rightarrow\text{-}E}{\neg \forall x.\, \neg A(x)}$$

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

$\exists$-*I* as a Derived Rule

The rule:

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

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

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

$\exists$-E as a Derived Rule

The rule:

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

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

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

$\exists$-E as a Derived Rule

The rule:

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

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

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

$\exists$-E as a Derived Rule

The rule:

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

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

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

We assume a hypothetical derivation.

$\exists$-E as a Derived Rule

The rule:

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

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

$$\dfrac{\neg B \qquad \begin{array}{c}A(x)\\ \vdots \\ B\end{array}}{\bot}\ \rightarrow\text{-}E$$

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

$\exists$-E as a Derived Rule

The rule:

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

$$\dfrac{\dfrac{\neg B \qquad \begin{array}{c}[A(x)]^2\\ \vdots \\ B\end{array}}{\bot}\ \to\text{-}E}{\neg A(x)}\ \to\text{-}I^2$$

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

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

$\exists$-E as a Derived Rule

The rule:

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

$$\neg\forall x.\, \neg A(x) \qquad \dfrac{\dfrac{\dfrac{\neg B \qquad \begin{array}{c}[A(x)]^2\\ \vdots \\ B\end{array}}{\bot}\ \rightarrow\text{-}E}{\neg A(x)}\ \rightarrow\text{-}I^2}{\forall x.\, \neg A(x)}\ \forall\text{-}I$$

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

$\exists$-E as a Derived Rule

The rule:

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

$$\dfrac{\dfrac{\dfrac{\dfrac{\neg B \qquad \begin{array}{c}[A(x)]^2\\ \vdots \\ B\end{array}}{\bot}\ \rightarrow\text{-}E}{\neg A(x)}\ \rightarrow\text{-}I^2}{\forall x.\, \neg A(x)}\ \forall\text{-}I \qquad \neg\forall x.\, \neg A(x)}{\bot}\ \rightarrow\text{-}E$$

We apply $\rightarrow$-E.

$\exists$-E as a Derived Rule

The rule:

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

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

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

Sample Derivation

Assumption: x does not occur free in B

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

Conclusion on FOL

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

Conclusion on FOL

- Propositional logic is good for modeling simple patterns of reasoning like "if . . . then . . . else".
- In first-order logic, one has "elements in a universe of discourse" and relations on / properties of them. One can quantify over the elements of the universe. Powerful!
- Some people advocate intuitionistic, relevant, and other "deviant" logics.
- Limitation: cannot quantify over predicates.
- "A" world or "the" world is modeled in first-order logic using so-called first-order theories. This will be studied

next lecture.

More Detailed Explanations

Boolean Functions

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

Relations/Functions, Infinity

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

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

What is a Domain?

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

Any set of elements of the universe of discourse that we want to reason about.

Alternating State

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

The State Alternates over all Time

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

x is some property which may or may not hold at a given point of time t, expressed as a number. So we write $x(t)$ if x holds at t. The formula says that x is true at point t if and only if x is not true at point $t+1$.

To be more precise, one would have to axiomatize the fact that t is a number etc.

"Constant Elements"?

As opposed to a variable which ranges over all values (elements of the universe of discourse).

Function Notation

So a function symbol f denotes an operation that takes n elements of the universe of discourse and returns a another one: $f(t_1, \ldots, t_n)$ is an element that depends on $t_1, \ldots, t_n$.

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

Equational Axiomatization

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

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

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

"Some values"?

Just like a constant, a variable stands for a value (element of discourse).

The most important difference between a constant and a (free) variable is that the latter ranges over all values.

Variables can be bound by quantifiers, so one can make statements such as "for all x . . . " or "there exists x such that . . . ".

What is Satisfiability? Validity?

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

More formally, this will be defined later.

Syntactic Categories

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

A term is an expression that stands for an element in the universe of discourse called a value.

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

But couldn't a statement also be a value? And couldn't a value depend on a statement?

In first-order logic the answer is: no!

Signatures

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

One often calls the pair $(\mathcal{F}, \mathcal{P})$ a signature. Generally, a signature specifies the "fixed symbols" (as opposed to variables) of a particular logic language.

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

these symbols is not relevant.

A Language

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

Free, Bound, and Binding Occurrences

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

1. The (only) occurrence of x in the term x is a free occurrence of x in x;
2. the free occurrences of x in $f(t_1, \ldots, t_n)$ 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.

We also define

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

Structures

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

A universe (of discourse) is sometimes also called domain.

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

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

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

symbols;

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

As before, we assume that the signature is clear from the context.

Strictly speaking, we should say "structure for a particular signature".

Details can be found in any textbook on logic [vD80].

Omitted Cases

The following cases have been omitted:

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

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

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

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

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

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

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

Models

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

As explained before, it is common to distinguish an interpretation, which gives a meaning to the constant 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 or formula. In the alternative terminology, we write $\mathcal{I}(A)(.)$ instead. This makes sense since in the alternative terminology, $\mathcal{I}$ and A together contain the same information as $\mathcal{A}$ in the original terminology. We define:

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

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

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

Suitable Structures

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

$\mathcal{N}$

$\mathcal{N}$ denotes the natural numbers.

Confusion of Syntax and Semantics?

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

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

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

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

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

Why is this a Model?

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

Why is this not a Model?

The identity function maps every object to itself.

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

Implicit Quantifiers

In the statement

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

the $\forall$-quantifier is implicit. It should be

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

Adapt the Proof for Reals

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

A proof for the reals:
Consider an arbitrary x where $x > 2$. Then $x = 2 + y$ for some $y > 0$ and hence

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

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

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

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

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

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

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

Inheriting Rules

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

Schematic Rules

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

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

Reflexivity

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

Side Condition Violated!

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

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

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

Substitutions in FOL

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

1. $x[x \leftarrow t] = t$;
2. $y[x \leftarrow t] = y$ if y is a variable other than x;
3. $f(t_1, \ldots, t_n)[x \leftarrow t] = f(t_1[x \leftarrow t], \ldots, t_n[x \leftarrow t])$ (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 $\perp$);
5. $(\neg\psi)[x \leftarrow t] = \neg(\psi[x \leftarrow t])$
6. $(\psi \circ \phi)[x \leftarrow t] = (\psi[x \leftarrow t] \circ \phi[x \leftarrow t])$ (where $\circ \in \{\wedge, \vee, \rightarrow\}$);
7. $(\mathsf{Q}x.\psi)[x \leftarrow t] = \mathsf{Q}x.\psi$ (where $\mathsf{Q} \in \{\forall, \exists\}$);

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 $z \notin FV(\psi)$.

Avoiding Capture of Variables

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

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

The definition can be found in any textbook on logic [vD80]. We will also give a formal definition later, in the context of the λ-calculus.

Check Side Conditions

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

Defining $\leftrightarrow$

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

Defining $\exists$

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

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

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

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

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

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

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

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

Hypothetical Derivation

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

Note two points:

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

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

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

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

Inheriting a Side Condition

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

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

Classical Reasoning

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

The Power of First-Order Logic

In first-order logic, one has values(elements of the universe of discourse) and relations/properties that may or may not hold for these values. Quantifiers are used to speak about "all values" and "some value".

For example, one can reason:

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

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

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

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

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

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

In English, the example is commonly given using the word "man", although one also finds "human". Like many languages (e.g., French, Italian), English often uses "man" for "human being", although this use of language may be considered discriminating against women.
E.g. [Tho95a]:

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

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

While the example does not, strictly speaking, imply that "man" is used in the meaning of "human being", this is strongly suggested both by the content of the example (or should women be immortal?) and the fact that languages that do have a word for "human being" (e.g. "Mensch" in German) usually give the example using this word. In fact, the example is originally in Old Greek, and there the word ἄνθρωπος (anthropos = human being), as opposed to 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 frequently uses Socrates and Kallias as standard names for individuals [Ari]. Possibly there were statutes of Socrates and Kallias standing in the hall where Aristotle gave his lectures, so it was convenient for him to point to the statutes whenever he was making a point involving two individuals.

Other Logics

There are still controversies about what the best logic is for reasoning about values (=elements of a universe of discourse) and properties/relations, and scope (quantification). Some argue for intuitionistic, relevance, modal and other "deviant" logics.

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

> From "a dollar buys a candy bar" and "a dollar buys an ice cream" we cannot normally conclude "a dollar buys a candy bar and an ice cream".

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

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

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

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

Limitations of First-Order Logic

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

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

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

since we would be quantifying over p.

First-Order Logic: Theories

David Basin, Burkhart Wolff, and Jan-Georg Smaus

Overview

Last lecture: first-order logic.

This lecture:

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

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

FOL with Equality

Equality is a logical symbol rather than a mathematical one.

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

Syntax and Semantics

Syntax: $=$ is a binary infix predicate.

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

Syntax and Semantics

Syntax: $=$ is a binary infix predicate.

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

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

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

Note the three completely different uses of "$=$" here!

Rules

- Equality is an equivalence relation

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

Rules

- Equality is an equivalence relation

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

- Equality is also a congruence on terms and all relations

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

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

Congruence: Alternatives

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

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

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

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

Examples

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

Examples

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

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

Examples

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

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

We show two ways:

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

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

Isabelle Rule

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

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

or literally

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

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

$$\dfrac{\dfrac{}{t = t}\,\textit{refl}}{\exists x.\, t = x}\,\exists\text{-}I$$

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

$$\dfrac{\dfrac{}{t = t}\,\textit{refl}}{\exists x.\, t = x}\,\exists\text{-}I$$

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

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

$$\dfrac{\dfrac{}{t = t}\,\textit{refl}}{\exists x.\, t = x}\,\exists\text{-}I$$

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

Notational confusion avoided by a precise metalanguage.

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

$$\dfrac{\dfrac{}{t = t}\,\textit{refl}}{\exists x.\, t = x}\,\exists\text{-}I$$

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

Notational confusion avoided by a precise metalanguage.

More Detailed Explanations

Logical vs. Non-logical Symbols

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

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

What status should the equality symbol $=$ have? We will assume that $=$ is a symbol whose meaning is hard-wired into the formalism. One then speaks of first-order logic with equality.

Three Different Uses of Equality

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

The first $=$ is a predicate symbol.

Three Different Uses of Equality

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

The first $=$ is a predicate symbol.

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

Three Different Uses of Equality

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

The first $=$ is a predicate symbol.

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

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

Why Rules?

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

What is an Equivalence?

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

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

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

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

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

What is a Congruence?

In general mathematical terminology, a relation $\cong$ is a congruence w.r.t. (or: on) f, where f has arity n, if $a_1 \cong b_1, \ldots, a_n \cong b_n$ 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 $14 \cdot 15 \equiv 8 \cdot 9 \bmod 6$.

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.

$15 \equiv 9 \bmod 6$ and 15 is divisible by 3, hence 9 is divisible by 3.

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

Soundness of Equivalence Rules

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

Semantic equality is an equivalence relation.

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

Soundness of Congruence Rules

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

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

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

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

Occurrences vs. Substitution

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

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

Substitutions are a way to make the notion of variable occurence precise: assume we have a term t containing the (free) variable x.

Now, we can represent the n occurences of x in t by terms t_1 to t_n, for which $t = t_i[z \longleftarrow x]$ holds, where z is not a free variable in t and where z appears only once in the term. Then, the t_i represents an occurrence of x in t which is marked by z.

See also example.

Example: $x^2 + y^2 > 12 \cdot x$

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

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

Now the atom above the line in the examples is obtained by substituting x for z, and the atom below the line is obtained by substituting y for z.

The Substitutivity Rule

The FOL rule for Substitutivity ("Leipnitz Rule") is presented as:

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

We can think of $P(x)$ and $P(y)$ as $P[z \longleftarrow x]$ for some arbitrary z. Think of P as a formula where some positions are marked in such a way that once we apply P to t (we write $P(t)$), t will be substituted into all those positions.

In fact, the particular choice of z does not play a role; it is an "anonymous" variable from the point of view of substitutivity. This motivates the λ-calculus which allows for writing $\lambda z.P$ for this situation.

Why Are All Functions in a Structure Total?

If we allowed partial functions in a structure, then terms t can be undefined, and elementary operations like substitution require all sorts of side-conditions (we must not replace a variable with an undefined term, etc.).

First-Order Theories

David Basin, Burkhart Wolff, and Jan-Georg Smaus

Example 1: Partial Orders

- The language of the theory of partial orders: $\leq$

Example 1: Partial Orders

- The language of the theory of partial orders: $\leq$
- Axioms

$$\forall x, y, z.\, x \leq y \wedge y \leq z \rightarrow x \leq z$$
$$\forall x, y.\, x \leq y \wedge y \leq x \leftrightarrow x = y$$

Example 1: Partial Orders

- The language of the theory of partial orders: $\leq$
- Axioms

$$\forall x, y, z.\, x \leq y \wedge y \leq z \rightarrow x \leq z$$
$$\forall x, y.\, x \leq y \wedge y \leq x \leftrightarrow x = y$$

- Alternative to axioms is to convert to rules

$$\frac{x \leq y \quad y \leq z}{x \leq z}\ \textit{trans} \qquad \frac{x \leq y \quad y \leq x}{x = y}\ \textit{antisym} \qquad \frac{x = y}{x \leq y}\ \textit{\leq-refl}$$

Such a conversion is possible since implication is the main connective.

A Second Transitivity Rule

One may also consider adding the rule

$$\frac{x = y}{y \leq x}\ \leq\text{-}\mathit{refl2}$$

to the system. This rule can be derived as follows:

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

More on Orders

- A partial order is a linear or total order when

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

Note: no "pure" rule formulation of this disjunction.

More on Orders

- A partial order is a linear or total order when

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

  Note: no "pure" rule formulation of this disjunction.

- A total order is dense when, in addition

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

  What does $<$ mean?

Structures for Orders . . .

Give structures for orders that are . . .

1. partial but not total:

Structures for Orders . . .

Give structures for orders that are . . .

1. partial but not total: $\subseteq$-relation;
2. total but not dense:

Structures for Orders . . .

Give structures for orders that are . . .

1. partial but not total: $\subseteq$-relation;
2. total but not dense: integers with $\leq$;
3. dense:

Structures for Orders . . .

Give structures for orders that are . . .

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

Example 2: Groups

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

Example 2: Groups

- Language: Function symbols $_ \cdot _$, $_^{-1}$, e
- A group is a model of

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

It is an example of an equational theory.

Example 2: Groups

- Language: Function symbols $_ \cdot _$, $_^{-1}$, e
- A group is a model of

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

It is an example of an equational theory.

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

Theorem 1

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

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

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

Theorem 1

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

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

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

Theorem 1

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

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

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

Theorem 1

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

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

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

Theorem 1

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

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

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

Theorem 1

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

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

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

Theorem 1

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

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

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

Theorem 1

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

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

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

Theorem 1

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

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

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

Theorem 1

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

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

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

Theorem 1

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

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

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

Theorem 1

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

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

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

Theorem 1

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

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

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

Theorem 1

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

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

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

Theorem 1

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

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

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

Theorem 2

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

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

$e \cdot x$

Theorem 2

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

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

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

Theorem 2

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

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

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

Theorem 2

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

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

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

Theorem 2

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

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

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

Theorem 2

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

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

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

Theorem 2

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

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

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

Theorem 2

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

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

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

Theorem 2

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

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

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

Lessons Learned from these Examples

Equational proofs are often tricky!

- Equalities used in different directions, "eureka" terms, etc.
- In some cases (the word problem is) decidable.

Equational versus ND Proofs

- Above proofs were of a particular, equational form.

Equational versus ND Proofs

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

  Term rewriting is a process for replacing equals by equals (see later).
- Alternative is natural deduction:
  - requires explicit proofs using equality rules;
  - tedious in practice. Try it on above examples!

More Detailed Explanations

Theories

Recall our intuitive explanation of theories.

A theory involves certain function and/or predicate symbols for which certain "laws" hold.

Depending on the context, these symbols may co-exist with other symbols.

Technically, the laws are added as rules (in particular, axioms) to the proof system.

A structure in which these rules are true is then called a model of the rules.

Partial Orders

A partial order is a binary relation that is reflexive, transitive, and anti-symmetric: $a \leq b$ and $b \leq a$ implies $a = b$.

A Language Consisting of $\leq$?

$\leq$ is (by convention) a binary infix predicate symbol.

The theory of partial orders involves only this symbol, but that does not mean that there could not be any other symbols in the context.

Antisymmetry and Reflexivity

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

Transitivity

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

Axioms vs. Rules

One can see that using $\rightarrow$-*I* and $\rightarrow$-*E*, one can always convert a proof using the axioms to one using the proper rules.

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

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

Do it in Isabelle!

Linear and Dense Orders

We define these notions in a usual mathematical terminology.

A partial order $\leq$ is linear or total if for all a, b, either $a \leq b$ or $b \leq a$.

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

"Pure" Rule Formulation

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

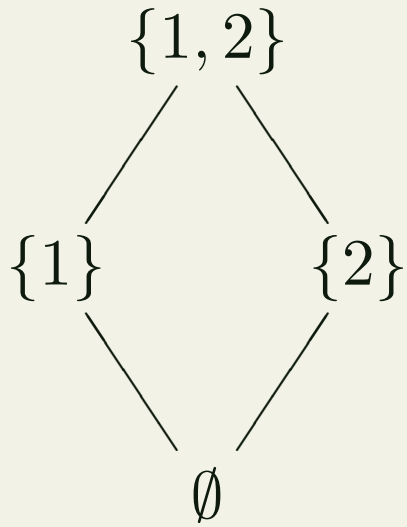
$<$

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

We say that $<$ is the strict part of the partial order $\leq$.

The $\subseteq$-Relation

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



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

In this example, we have the partial order

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

Group Language

$_ \cdot _$ is a binary infix function symbol (in fact, only $\cdot$ is the symbol, but the notation $_ \cdot _$ is used to indicate the fact that the symbol stands between its arguments).

$_^{-1}$ is a unary function symbol written as superscript. Again, the $_$ is used to indicate where the argument goes.

e is a nullary function symbol (= constant).

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

Group

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

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

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

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

Equational Theory

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

Sometimes, each equation is surrounded by several $\forall$-quantifiers binding all the free variables in the equation, but often the equation is regarded as implicitly universally quantified.

More generally, a conditional equational theory consists of proper rules where the premises are called conditions [Höl90].

Note also that sometimes, one also considers the basic rules of equality as being part of every equational theory. Whenever one has an equational theory, one implies that the basic rules are present; whether or not one assumes that they are formally elements of the equational theory is just a technical detail.

A Model a Group?

A model of the group axioms is a structure in which the group axioms are true.

However, when we say something like, "this model is a group", then this is a slight abuse of terminology, since there may be other function symbols around that are also interpreted by the structure.

So when we say "this model is a group", we mean, "this model is a model of the group axioms for function symbols $_ \cdot _$, $_^{-1}$,and e clear from the context".

"Eureka" terms

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

The Greek heureka is 1st person singular perfect of heuriskō, "to find". It was exclaimed by Archimedes upon discovering how to test the purity of Hiero's crown.

The Word Problem

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

Equational Proofs

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

This style of proof can be justified by the rules given for equality, in particular the congruences. However, it looks very different from the natural deduction style.

Proof of Theorem 2 by Natural Deduction

$$\frac{}{e \cdot x = x}$$

Most steps use the congruence rule $cong_2$.

Proof of Theorem 2 by Natural Deduction

$$\dfrac{\dfrac{}{x \cdot e = x} \qquad \dfrac{}{e \cdot x = x \cdot e}}{e \cdot x = x}$$

Most steps use the congruence rule $cong_2$.

Proof of Theorem 2 by Natural Deduction

$$\dfrac{\dfrac{\boxed{\text{r-neutr}}}{x \cdot e = x} \qquad \dfrac{}{e \cdot x = x \cdot e}}{e \cdot x = x}$$

Most steps use the congruence rule $cong_2$.

Proof of Theorem 2 by Natural Deduction

$$\dfrac{\dfrac{\boxed{\text{r-neutr}}}{x \cdot e = x} \qquad \dfrac{\dfrac{}{x^{-1} \cdot x = e} \qquad \dfrac{}{e \cdot x = x \cdot (x^{-1} \cdot x)}}{e \cdot x = x \cdot e}}{e \cdot x = x}$$

Most steps use the congruence rule $cong_2$.

Proof of Theorem 2 by Natural Deduction

$$\dfrac{\dfrac{\boxed{\text{r-neutr}}}{x \cdot e = x} \qquad \dfrac{\dfrac{\text{Theorem 1}}{x^{-1} \cdot x = e} \qquad \dfrac{}{e \cdot x = x \cdot (x^{-1} \cdot x)}}{e \cdot x = x \cdot e}}{e \cdot x = x}$$

Most steps use the congruence rule $cong_2$.

Proof of Theorem 2 by Natural Deduction

$$
\dfrac{\dfrac{\boxed{\text{r-neutr}}}{x \cdot e = x} \qquad \dfrac{\dfrac{\text{Theorem 1}}{x^{-1} \cdot x = e} \qquad \dfrac{\dfrac{}{(x \cdot x^{-1}) \cdot x = x \cdot (x^{-1} \cdot x)} \qquad \dfrac{}{e \cdot x = (x \cdot x^{-1}) \cdot x}}{e \cdot x = x \cdot (x^{-1} \cdot x)}}{e \cdot x = x \cdot e}}{e \cdot x = x}
$$

Most steps use the congruence rule $cong_2$.

Proof of Theorem 2 by Natural Deduction

$$
\dfrac{\dfrac{\boxed{\text{r-neutr}}}{x \cdot e = x} \qquad \dfrac{\dfrac{\text{Theorem 1}}{x^{-1} \cdot x = e} \qquad \dfrac{\dfrac{\boxed{\text{assoc}}}{(x \cdot x^{-1}) \cdot x = x \cdot (x^{-1} \cdot x)} \qquad \dfrac{}{e \cdot x = (x \cdot x^{-1}) \cdot x}}{e \cdot x = x \cdot (x^{-1} \cdot x)}}{e \cdot x = x \cdot e}}{e \cdot x = x}
$$

Most steps use the congruence rule $cong_2$.

Proof of Theorem 2 by Natural Deduction

$$
\dfrac{\dfrac{\boxed{\text{r-neutr}}}{x \cdot e = x} \qquad \dfrac{\dfrac{\text{Theorem 1}}{x^{-1} \cdot x = e} \qquad \dfrac{\dfrac{\boxed{\text{assoc}}}{(x \cdot x^{-1}) \cdot x = x \cdot (x^{-1} \cdot x)} \qquad \dfrac{\text{cont. below}}{e \cdot x = (x \cdot x^{-1}) \cdot x}}{e \cdot x = x \cdot (x^{-1} \cdot x)}}{e \cdot x = x \cdot e}}{e \cdot x = x}
$$

Most steps use the congruence rule $cong_2$.

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

$$\dfrac{\dfrac{}{e = x \cdot x^{-1}}\,\textit{sym} \qquad \dfrac{}{e \cdot x = e \cdot x}\,\textit{refl}}{e \cdot x = \left(x \cdot x^{-1}\right) \cdot x}$$

$$\dfrac{\dfrac{\dfrac{\boxed{\text{r-inv}}}{x \cdot x^{-1} = e}}{e = x \cdot x^{-1}}\,\textit{sym} \qquad \dfrac{}{e \cdot x = e \cdot x}\,\textit{refl}}{e \cdot x = \left(x \cdot x^{-1}\right) \cdot x}$$

Proof of Theorem 2 by Natural Deduction, Complete

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

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

Naïve Set Theory

David Basin, Burkhart Wolff, and Jan-Georg Smaus

Naïve Set Theory: Basics

- A set is a collection of objects where order and repetition are unimportant.

  Sets are central in mathematical reasoning [Vel94]. E.g., set of prime numbers.

Naïve Set Theory: Basics

- A set is a collection of objects where order and repetition are unimportant.

  Sets are central in mathematical reasoning [Vel94]. E.g., set of prime numbers.
- In what follows we consider a simple, intuitive formalization: "naïve set theory".

Naïve Set Theory: Basics

- A set is a collection of objects where order and repetition are unimportant.

  Sets are central in mathematical reasoning [Vel94]. E.g., set of prime numbers.

- In what follows we consider a simple, intuitive formalization: "naïve set theory".

  We will be somewhat less formal than usual. Our goal is to understand standard mathematical practice.

Naïve Set Theory: Basics

- A set is a collection of objects where order and repetition are unimportant.

  Sets are central in mathematical reasoning [Vel94]. E.g., set of prime numbers.

- In what follows we consider a simple, intuitive formalization: "naïve set theory".

  We will be somewhat less formal than usual. Our goal is to understand standard mathematical practice.

  Later, in HOL, we will be completely formal.

Sets: Language

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

- set-comprehension $\{x|P(x)\}$ and a binary membership predicate $\in$.

Sets: Language

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

- set-comprehension $\{x|P(x)\}$ and a binary membership predicate $\in$.
- Term/formula distinction inadequate: need a syntactic category for sets.

Sets: Language

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

- set-comprehension $\{x|P(x)\}$ and a binary membership predicate $\in$.
- Term/formula distinction inadequate: need a syntactic category for sets.
- We will be more formal about syntax later (HOL).

Sets: Language

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

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

Examples

- $\forall x.\, x \in \{y | y \bmod 6 = 0\} \rightarrow (x \bmod 2 = 0 \wedge x \bmod 3 = 0)$.

Examples

- $\forall x.\, x \in \{y | y \bmod 6 = 0\} \rightarrow (x \bmod 2 = 0 \wedge x \bmod 3 = 0)$.
- What does the following say?

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

Examples

- $\forall x.\, x \in \{y | y \bmod 6 = 0\} \rightarrow (x \bmod 2 = 0 \wedge x \bmod 3 = 0)$.
- What does the following say?

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

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

Proof Rules for Sets

Introduction, elimination, extensional equality

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

$$\frac{\forall x.\, x \in A \leftrightarrow x \in B}{A = B} \qquad \frac{A = B}{\forall x.\, x \in A \leftrightarrow x \in B}$$

Following equivalence is derivable:

$$\forall x.\, P(x) \leftrightarrow x \in \{y|P(y)\}$$

Digression: Sorted Reasoning

- In mathematical arguments we often (implicitly) assume that variables are restricted to some universe of discourse. E.g., $x^2 < 9$ (universe either $\mathcal{R}$, $\mathcal{N}$, . . .)
- To avoid ambiguity we can include sort information in formulae:

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

Formally

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

Sorted Reasoning in an Unsorted Logic

- We may introduce the additional set comprehension syntax $\{x \in A | P(x)\}$, but our logic is still unsorted. We have

$$y \in \{x \in A | P(x)\} \leftrightarrow y \in \{x \mid A(x) \wedge P(x)\} \leftrightarrow A(y) \wedge P(y)$$

- Sorted quantification

$$\begin{aligned} \forall x \in A.\, P(x) &\equiv \forall x.\, A(x) \rightarrow P(x) \\ \exists x \in A.\, P(x) &\equiv \exists x.\, A(x) \wedge P(x) \end{aligned}$$

Operations on Sets

- Functions on sets

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

- Predicates on sets

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

Examples of Operations on Sets

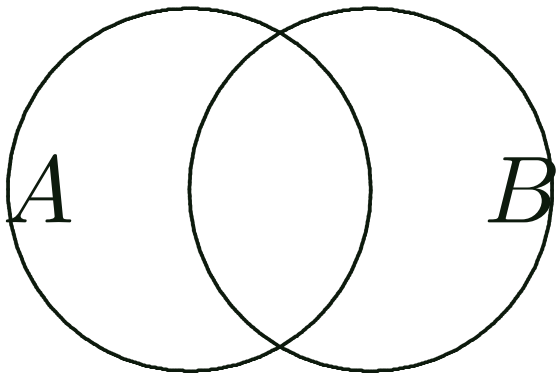
One often depicts sets as circles or bubbles.

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

Examples of Operations on Sets

One often depicts sets as circles or bubbles.

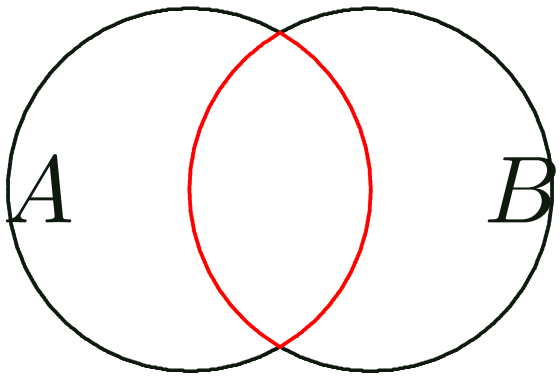
What are $A \cap B$, $A \cup B$, $A \setminus B$?



Examples of Operations on Sets

One often depicts sets as circles or bubbles.

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

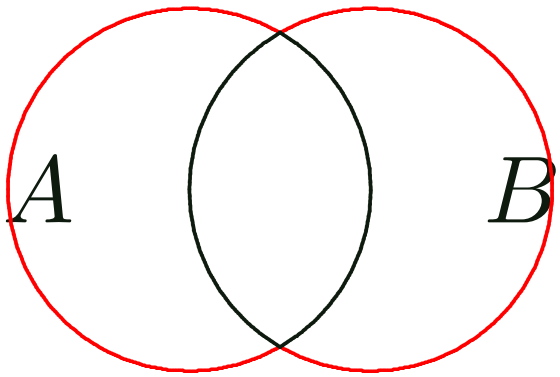


$A \cap B$

Examples of Operations on Sets

One often depicts sets as circles or bubbles.

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

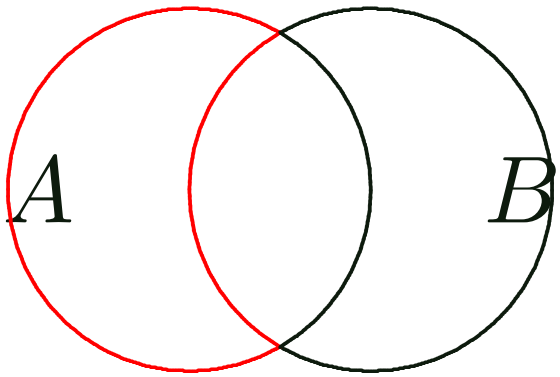


$A \cup B$

Examples of Operations on Sets

One often depicts sets as circles or bubbles.

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



$A \setminus B$

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 and $\forall x.\, P(x) \leftrightarrow x \in \{y | P(y)\}$.

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 and $\forall x.\, P(x) \leftrightarrow x \in \{y | P(y)\}$.

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

Correspondence between Set-Theoretic and Logical Operators

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

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

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

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

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

Venn diagram (Is this a proof?)

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

Natural deduction (refinement style, natural language)

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

Natural deduction (refinement style, natural language)

By extensionality, suffices to show

$$\forall x.\, x \in A \cap (B \cup C) \leftrightarrow x \in (A \cap B) \cup (A \cap C).$$

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

Natural deduction (refinement style, natural language)

By extensionality, suffices to show

$$\forall x.\, x \in A \cap (B \cup C) \leftrightarrow x \in (A \cap B) \cup (A \cap C).$$

For an arbitrary x, this is equivalent to establishing

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

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

Natural deduction (refinement style, natural language)

By extensionality, suffices to show

$$\forall x.\, x \in A \cap (B \cup C) \leftrightarrow x \in (A \cap B) \cup (A \cap C).$$

For an arbitrary x, this is equivalent to establishing

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

But that is a propositional tautology.

Same in Isabelle

Last proof carries over to Isabelle: extensionality, rewriting, tautology checking. Do it!

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

Not obvious? Just follow your nose!

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

Not obvious? Just follow your nose!

Let A and B be arbitrary sets.

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

Not obvious? Just follow your nose!

Let A and B be arbitrary sets.

Let x be element of $(A \cup B) \setminus B$.

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

Not obvious? Just follow your nose!

Let A and B be arbitrary sets.
Let x be element of $(A \cup B) \setminus B$.
So $(x \in A \vee x \in B) \wedge \neg x \in B$.

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

Not obvious? Just follow your nose!

Let A and B be arbitrary sets.
Let x be element of $(A \cup B) \setminus B$.
So $(x \in A \vee x \in B) \wedge \neg x \in B$.
Therefore $x \in A$.

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

Not obvious? Just follow your nose!

Let A and B be arbitrary sets.
Let x be element of $(A \cup B) \setminus B$.
So $(x \in A \vee x \in B) \wedge \neg x \in B$.
Therefore $x \in A$.
Therefore $x \in (A \cup B) \setminus B \rightarrow x \in A$.

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

Not obvious? Just follow your nose!

Let A and B be arbitrary sets.
Let x be element of $(A \cup B) \setminus B$.
So $(x \in A \vee x \in B) \wedge \neg x \in B$.
Therefore $x \in A$.
Therefore $x \in (A \cup B) \setminus B \rightarrow x \in A$.
Therefore $((A \cup B) \setminus B) \subseteq A$.

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

Not obvious? Just follow your nose!

Let A and B be arbitrary sets.
Let x be element of $(A \cup B) \setminus B$.
So $(x \in A \vee x \in B) \wedge \neg x \in B$.
Therefore $x \in A$.
Therefore $x \in (A \cup B) \setminus B \rightarrow x \in A$.
Therefore $((A \cup B) \setminus B) \subseteq A$.

This semi-formal proof combines forward reasoning with backward reasoning. This is common in practice and usually easy to unscramble.

Extending Set Comprehensions

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

Extending Set Comprehensions

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

Can define set transformers, e.g.,

$$\{f(x)|P(x)\}$$

Extending Set Comprehensions

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

Can define set transformers, e.g.,

$$\{f(x)|P(x)\} \equiv \{y|\exists x.\, P(x) \wedge y = f(x)\}$$

Extending Set Comprehensions

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

Can define set transformers, e.g.,

$$\{f(x)|P(x)\} \equiv \{y|\exists x.\, P(x) \wedge y = f(x)\}$$

Example: $t \in \{x^2|x > 5\}$ equivalent to

Extending Set Comprehensions

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

Can define set transformers, e.g.,

$$\{f(x)|P(x)\} \equiv \{y|\exists x.\, P(x) \wedge y = f(x)\}$$

Example: $t \in \{x^2|x > 5\}$ equivalent to $\exists x.\, x > 5 \wedge t = x^2$.

True for $t \in \{36, 49, \ldots\}$

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 $f(x)$.

Indexing

Sometimes, it is natural to denote a function f applied to an argument x as "f indexed by x", so f_x, rather than $f(x)$.

Example: let S = set of students and let m_s stand for "the mother of s", for s a student. Call S an index set.

$$\begin{aligned} x \in \{m_s | s \in S\} &\leftrightarrow x \in \{y | \exists s.\, s \in S \wedge y = m_s\} \\ &\leftrightarrow \exists s.\, s \in S \wedge x = m_s \\ &\leftrightarrow \exists s \in S.\, x = m_s \end{aligned}$$

Indexing

Sometimes, it is natural to denote a function f applied to an argument x as "f indexed by x", so f_x, rather than $f(x)$.

Example: let S = set of students and let m_s stand for "the mother of s", for s a student. Call S an index set.

$$\begin{aligned} x \in \{m_s | s \in S\} &\leftrightarrow x \in \{y | \exists s.\, s \in S \wedge y = m_s\} \\ &\leftrightarrow \exists s.\, s \in S \wedge x = m_s \\ &\leftrightarrow \exists s \in S.\, x = m_s \end{aligned}$$

Uses extended comprehensions, indexing syntax, and sorted quantification.

Logical Forms of the New Notation

Question: what is the logical form of $\{x_i | i \in I\} \subseteq A$?

Logical Forms of the New Notation

Question: what is the logical form of $\{x_i | i \in I\} \subseteq A$?

$$\forall x.\, x \in \{x_i | i \in I\} \rightarrow x \in A, \quad \text{i.e.,}$$
$$\forall x.\, (\exists i \in I.\, x = x_i) \rightarrow x \in A.$$

Logical Forms of the New Notation

Question: what is the logical form of $\{x_i | i \in I\} \subseteq A$?

$$\forall x.\, x \in \{x_i | i \in I\} \rightarrow x \in A, \quad \text{i.e.,}$$
$$\forall x.\, (\exists i \in I.\, x = x_i) \rightarrow x \in A.$$

Intuition suggests that $\forall i \in I.\, x_i \in A$ is also correct, i.e.,

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

Can you prove this?

Indexed Families

Can formulate sets as indexed families.

Let S = set of students, C_s = courses taken by student s.

Then

$$\{C_s | s \in S\}$$

Indexed Families

Can formulate sets as indexed families.

Let S = set of students, C_s = courses taken by student s. Then

$$\{C_s | s \in S\}$$

is the set whose elements are those sets of courses taken by some student.

Logical Forms of Powersets

$$\mathcal{P}(A) = \{x | x \subseteq A\}.$$

What is the logical form of:

1. $x \in \mathcal{P}(A)$?

Logical Forms of Powersets

$$\mathcal{P}(A) = \{x | x \subseteq A\}.$$

What is the logical form of:

1. $x \in \mathcal{P}(A)$?

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

2. $\mathcal{P}(A) \subseteq \mathcal{P}(B)$?

Logical Forms of Powersets

$$\mathcal{P}(A) = \{x | x \subseteq A\}.$$

What is the logical form of:

1. $x \in \mathcal{P}(A)$?

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

2. $\mathcal{P}(A) \subseteq \mathcal{P}(B)$?

   $\forall x. x \in \mathcal{P}(A) \rightarrow x \in \mathcal{P}(B)$, i.e.,

Logical Forms of Powersets

$$\mathcal{P}(A) = \{x | x \subseteq A\}.$$

What is the logical form of:

1. $x \in \mathcal{P}(A)$?

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

2. $\mathcal{P}(A) \subseteq \mathcal{P}(B)$?

   $\forall x. x \in \mathcal{P}(A) \rightarrow x \in \mathcal{P}(B)$, i.e.,

   $\forall x. x \subseteq A \rightarrow x \subseteq B$, i.e.,

Logical Forms of Powersets

$$\mathcal{P}(A) = \{x | x \subseteq A\}.$$

What is the logical form of:

1. $x \in \mathcal{P}(A)$?

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

2. $\mathcal{P}(A) \subseteq \mathcal{P}(B)$?

   $\forall x. x \in \mathcal{P}(A) \rightarrow x \in \mathcal{P}(B)$, i.e.,

   $\forall x. x \subseteq A \rightarrow x \subseteq B$, i.e.,

   $\forall x. (\forall y. y \in x \rightarrow y \in A) \rightarrow (\forall y. y \in x \rightarrow y \in B)$

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

Outlook

Sets can have other sets as elements.

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

Russell's Paradox

Suppose $U := \{x \mid \top\}$. Then $U \in U$.

Somewhat unusual, but no contradiction yet.

Russell's Paradox

Suppose $U := \{x \mid \top\}$. Then $U \in U$.

Somewhat unusual, but no contradiction yet.

Now split sets into two categories:

1. unusual sets like U that are elements of themselves, and
2. more typical sets that are not.

Russell's Paradox

Suppose $U := \{x \mid \top\}$. Then $U \in U$.

Somewhat unusual, 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 $R := \{A | A \notin A\}$.

Using logical form we derive: $\forall A. (A \in R \leftrightarrow A \notin A)$

Russell's Paradox

Suppose $U := \{x \mid \top\}$. Then $U \in U$.

Somewhat unusual, 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 $R := \{A | A \notin A\}$.

Using logical form we derive: $\forall A. (A \in R \leftrightarrow A \notin A)$

Substituting R for A ($\forall$-E) yields $R \in R \leftrightarrow R \notin R$, which is a logical contradiction.

Consequences

- Naïve Set Theory is nice and highly intuitive ...

Consequences

- Naïve Set Theory is nice and highly intuitive . . .
- . . . but inconsistent!

Consequences

- Naïve Set Theory is nice and highly intuitive ...
- ... but inconsistent!
- Axioms must be considered harmful:
  "The axiomatic method has the advantage of theft over honest labour" (Russel)

Consequences

- Naïve Set Theory is nice and highly intuitive ...
- ... but inconsistent!
- Axioms must be considered harmful:
  "The axiomatic method has the advantage of theft over honest labour" (Russel)
- New concepts to avoid inconsistency are needed: Types, Conservativity, ...

Where Do We Go from here?

In the sequel of the course, we will turn to the λ-calculus for three reasons:

- it is basis for a metalanguage to avoid notational confusion

Where Do We Go from here?

In the sequel of the course, we will turn to the λ-calculus for three reasons:

- it is basis for a metalanguage to avoid notational confusion
- it allows for a uniform representation of substitution, unification, Resolution and other deduction techniques

Where Do We Go from here?

In the sequel of the course, we will turn to the λ-calculus for three reasons:

- it is basis for a metalanguage to avoid notational confusion
- it allows for a uniform representation of substitution, unification, Resolution and other deduction techniques
- it is a foundation fot Higher-order Logic: a formalism for (among other things) non-naïve set theory.

More Detailed Explanations

Set Comprehension

Set comprehension is a way of defining sets through predicates.

$\{x|P(x)\}$ stands for the set of elements of the universe for which $P(x)$ (some formula usually containing x) holds.

Is a Set a Term?

It is more adequate to regard a set as a term than as a formula. A set is considered a value in a universe of discourse, not a relation over values. However, it is in fact possible to model relations inside set theory; therefore, the distinction is purely syntactical and not conceptual.

Extensional Equality

Two things are extensionally equal if they are "equal in their effects". Thus two sets are equal if they have the same members, regardless of their syntactic representation.

Note that extensional equality may be undecidable.

Deriving Equivalence for Comprehensions

$$
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{[P(x)]^1}{x \in \{y|P(y)\}}\,\textit{compr-I}}{P(x) \to x \in \{y|P(y)\}}\,\to\text{-}I^1 \qquad \dfrac{\dfrac{[x \in \{y|P(y)\}]^2}{P(x)}\,\textit{compr-E}}{x \in \{y|P(y)\} \to P(x)}\,\to\text{-}I^2}{P(x) \to x \in \{y|P(y)\} \wedge x \in \{y|P(y)\} \to P(x)}\,\wedge\text{-}I}{P(x) \leftrightarrow x \in \{y|P(y)\}}\,\text{iff}}{\forall x.\, P(x) \leftrightarrow x \in \{y|P(y)\}}\,\forall\text{-}I
$$

Rules $\wedge$-*I*, $\to$-*I*, $\forall$-*I* were defined in previous lectures. The step marked with iff is not a proof step in the technical sense. We only make the expansion of a shorthand notation explicit.

Universes

We already know what a universe or domain is. To interpret a particular language, we have a structure interpreting all function symbols as functions on the universe.

However, it is often adequate to subdivide the universe into several "sub-universes". Those are called sorts. Note that a sort is a set.

For example, in a usual mathematical context, one may distinguish $\mathcal{R}$ (the real numbers) and $\mathcal{N}$ (the natural numbers) to say that $\sqrt{x}$ requires x to be of sort $\mathcal{R}$ and $x!$ requires x to be of sort $\mathcal{N}$.

Avoiding Ambiguity

We want to make explicit the sort of the variable in question. So we do not want the set of all x such that $P(x)$ holds, but only the ones of the right sort, so the ones for which $x \in U$ (U being the sort/universe) holds.

Note there is a certain confusion here, since we write $x \in U$ in one place (so U should be a set) and $U(x)$ in another (so U should be a predicate. This confusion is deliberate and quite common. One can identify a set (sort) U with a unary predicate U such that $U(t)$ is interpreted as $True$ iff t is a member of U.

The whole expression $\{x \in U | P(x)\}$ is a special kind of syntax. Therefore, you must look at it as a whole: it makes no sense to see any meaning just in, say, the bit $x \in U$ in this expression. It is called set

comprehension, and it is defined by

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

Sorted Logic

In sorted logic, sorts are part of the syntax. So the signature contains a fixed set of sorts. For each constant, it is specified what its sort is. For each function symbol, it is specified what the sort of each argument is, and what the sort of the result is. For each predicate symbol, it is specified what the sort of each argument is.

Terms and formulas that do not respect the sorts are not well-formed, and so they are not assigned a meaning.

In contrast, our logic is unsorted. The special syntax we provide for sorted reasoning is just syntactic sugar, i.e., we use it as shorthand and since it has an intuitive reasoning, but it has no impact on how expressive our logic is.

For any formal language (programming language, logic, etc.), the term "syntactic sugar" refers to syntax that is provided for the sake of

readability and brevity, but which does not affect the expressiveness of the language.

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

Sorted Quantification

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

Set Functions

$\cap$ is called intersection.

$\cup$ is called union.

$\setminus$ is called set difference.

$\subseteq$ is called inclusion.

The Logical Form

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

Is a Venn Diagram a Proof?

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

A Venn diagram is not a proof in the sense defined earlier.

Moreover, it would not even be acceptable as a proof according to usual mathematical practice. If it is unknown whether two sets have a non-empty intersection, how are we supposed to draw them? Trying to make a case distinctions (drawing several diagrams depending on the cases) is error-prone.

Venn diagrams are useful for illustration purposes, but they are not proofs.

Natural Language

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

We can also do this in formal logic, too.

Explanations for each Step

Let A and B be arbitrary sets. ($\forall$-*I*)

Explanations for each Step

Let A and B be arbitrary sets. ($\forall$-*I*)

Let x be an element of $(A \cup B) \setminus B$ (temporary assumption)

Explanations for each Step

Let A and B be arbitrary sets.	($\forall$-*I*)
Let x be an element of $(A \cup B) \setminus B$	(temporary assumption)
So $(x \in A \vee x \in B) \wedge \neg x \in B$	(equivalent proposition)

Explanations for each Step

Let A and B be arbitrary sets.	($\forall$-*I*)
Let x be an element of $(A \cup B) \setminus B$	(temporary assumption)
So $(x \in A \vee x \in B) \wedge \neg x \in B$	(equivalent proposition)
Therefore $x \in A$	(P follows from $(P \vee Q) \wedge \neg Q$)

Explanations for each Step

Let A and B be arbitrary sets.	($\forall$-*I*)
Let x be an element of $(A \cup B) \setminus B$	(temporary assumption)
So $(x \in A \vee x \in B) \wedge \neg x \in B$	(equivalent proposition)
Therefore $x \in A$	(P follows from $(P \vee Q) \wedge \neg Q$)
Therefore $x \in (A \cup B) \setminus B \rightarrow x \in A$	($\rightarrow$-*I*)

Explanations for each Step

Let A and B be arbitrary sets.	($\forall$-*I*)
Let x be an element of $(A \cup B) \setminus B$	(temporary assumption)
So $(x \in A \vee x \in B) \wedge \neg x \in B$	(equivalent proposition)
Therefore $x \in A$	(P follows from $(P \vee Q) \wedge \neg Q$)
Therefore $x \in (A \cup B) \setminus B \rightarrow x \in A$	($\rightarrow$-*I*)
Therefore $((A \cup B) \setminus B) \subseteq A$	(def of $\subseteq$)

Explanations for each Step

Let A and B be arbitrary sets.	($\forall$-*I*)
Let x be an element of $(A \cup B) \setminus B$	(temporary assumption)
So $(x \in A \vee x \in B) \wedge \neg x \in B$	(equivalent proposition)
Therefore $x \in A$	(P follows from $(P \vee Q) \wedge \neg Q$)
Therefore $x \in (A \cup B) \setminus B \rightarrow x \in A$	($\rightarrow$-*I*)
Therefore $((A \cup B) \setminus B) \subseteq A$	(def of $\subseteq$)

Concerning forward and backwards reasoning, one may look at it as follows: we first construct the derivation step at the root of the proof tree ($\forall$-*I*), and then we jump to a leaf (by making the temporary assumption) and work downwards from there.

Definition of $\subseteq$

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

follows from the definition of $\subseteq$.

Details of Logical Form

We want to show

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

$x \in \{x_i \mid i \in I\}$	$\equiv$	(def. of notation)
$x \in \{y \mid \exists i.\, i \in I \wedge y = x_i\}$	$\equiv$	*compr-I*
$\exists i.\, i \in I \wedge x = x_i$	$\equiv$	(Sorted quantification)
$\exists i \in I.\, x = x_i$		

Intuition for Indexed Sets

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

Proof

Want to prove

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

Proof

Want to prove

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

- "$\rightarrow$"

  Let $i \in I$ be arbitrary. Now from assumption (for the instance x_i) we have $(\exists j \in I.\, x_i = x_j) \rightarrow x_i \in A$. But premise is true for $i = j$, so $x_i \in A$.

Proof

Want to prove

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

- "$\rightarrow$"

  Let $i \in I$ be arbitrary. Now from assumption (for the instance x_i) we have $(\exists j \in I.\, x_i = x_j) \rightarrow x_i \in A$. But premise is true for $i = j$, so $x_i \in A$.

- "$\leftarrow$"

  Let x be arbitrary and assume $\exists i \in I.\, x = x_i$. So for some $i \in I$, we have $x = x_i$. Now $\forall i \in I.\, x_i \in A$. Hence $x \in A$.

"$\rightarrow$" in more Detail: Want to prove

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

Proof

Want to prove

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

- "$\rightarrow$"

  Let $i \in I$ be arbitrary. Now from assumption (for the instance x_i) we have $(\exists j \in I.\, x_i = x_j) \rightarrow x_i \in A$. But premise is true for $i = j$, so $x_i \in A$.

- "$\leftarrow$"

  Let x be arbitrary and assume $\exists i \in I.\, x = x_i$. So for some $i \in I$, we have $x = x_i$. Now $\forall i \in I.\, x_i \in A$. Hence $x \in A$.

"$\rightarrow$" in more Detail: Want to prove

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

We show $\forall i \in I.\, x_i \in A$ assuming $\forall x.(\exists i \in I.\, x = x_i) \rightarrow x \in A$.

So we show that for arbitrary $i \in I$, assuming $\forall x.(\exists i \in I.\, x = x_i) \rightarrow x \in A$, we have $x_i \in A$. So let $i \in I$ be arbitrary.

Since we have $\forall x.(\exists i \in I.\, x = x_i) \rightarrow x \in A$, by rule $\forall$-*E* we can specialize to $(\exists j \in I.\, x_i = x_j) \rightarrow x_i \in A$. But premise $(\exists j \in I.\, x_i = x_j)$ is true for $i = j$, and so $x_i \in A$, which is what was to be proven.

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

"$\leftarrow$" in more Detail: Want to prove

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

We show $\forall i \in I.\, x_i \in A$ assuming $\forall x.(\exists i \in I.\, x = x_i) \rightarrow x \in A$.
So we show that for arbitrary $i \in I$, assuming $\forall x.(\exists i \in I.\, x = x_i) \rightarrow x \in A$, we have $x_i \in A$. So let $i \in I$ be arbitrary.
Since we have $\forall x.(\exists i \in I.\, x = x_i) \rightarrow x \in A$, by rule $\forall$-*E* we can specialize to $(\exists j \in I.\, x_i = x_j) \rightarrow x_i \in A$. But premise $(\exists j \in I.\, x_i = x_j)$ is true for $i = j$, and so $x_i \in A$, which is what was to be proven.
This proof could be made more formal by drawing a proof tree or using Isabelle.
"$\leftarrow$" in more Detail: Want to prove

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

We show $\forall x.(\exists i \in I.\, x = x_i) \rightarrow x \in A$, assuming $\forall i \in I.\, x_i \in A$.
So we show that for arbitrary x, assuming $\forall i \in I.\, x_i \in A$, we have

$(\exists i \in I.\, x = x_i) \rightarrow x \in A$. So let x be arbitrary.

To show $(\exists i \in I.\, x = x_i) \rightarrow x \in A$, assume $\exists i \in I.\, x = x_i$. So for some $i \in I$, we have $x = x_i$. Now by our earlier assumption $\forall i \in I.\, x_i \in A$, and so it follows that $x \in A$. thus we have shown $x \in A$ under the assumption $(\exists i \in I.\, x = x_i)$, thus we have shown $(\exists i \in I.\, x = x_i) \rightarrow x \in A$, which is what was to be proven.

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

Families

The word family is sometimes used for a function that maps elements of an index-set (e.g. natural numbers) to sets.

Collections and Sets

We speak of collection of all sets in order to avoid a definitional circle (this is the traditional way to proceed).

In practice, we have "sets of sets" in set theory, and even "sets of all sets", which will lead to certain problems. . .

Logical Characterization

Recall $R := \{A | A \notin A\}$ and recall the notion of logical form.

Let A be arbitrary (for the formal reasoning applied here, arbitrary means: it could be a set, a number, a dog, the pope, anything whatsoever).

By the rules for set comprehension, we can prove $A \in \{A | A \notin A\} \rightarrow A \notin A$ and $A \notin A \rightarrow A \in \{A | A \notin A\}$, and so by definition of $\leftrightarrow$, we have $A \in R \leftrightarrow A \notin A$, and since A was arbitrary, by $\forall$-*I*, we have $\forall A. (A \in R \leftrightarrow A \notin A)$.

What does this Tell us about Sets?

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

The fundamental flaw of naïve set theory is that sets and predicates are arbitrarily mutual dependent. Ways out of this dilemma are:

1. constraining the comprehension on a hierarchy of sets ($\longrightarrow$ Zermelo-Fränkel-Set-Theory),
2. typing set expressions and ruling out "circular" constructs such as $x \in x$ ($\longrightarrow$ Higher-order Logic), or
3. contraining the mutual dependencies to "monotonic" ones; sets can be defined via sets if the result "grows", which rules out the $\neg$ in Russels antinomy ($\longrightarrow$ Domain Theory).

True

Assume that $\top$ is syntactic sugar for a proposition that is always true, say $\top \equiv \bot \rightarrow \bot$. We have not introduced this, but it is convenient.

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

A Strange Set Comprehension

Recall that a set comprehension has the form $\{x|P(x)\}$, where $P(x)$ is a formula usually containing x.

The set comprehension $U := \{x \mid \top\}$ is strange since $\top$ does not contain x.

But by the introduction rule for set comprehensions, this means that $x \in U$ for any x. Thus in particular, $U \in U$.

Higher-Order Logic

Higher-order logic is a solution to the dilemma presented by Russell's paradox.

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

Motivation and Background

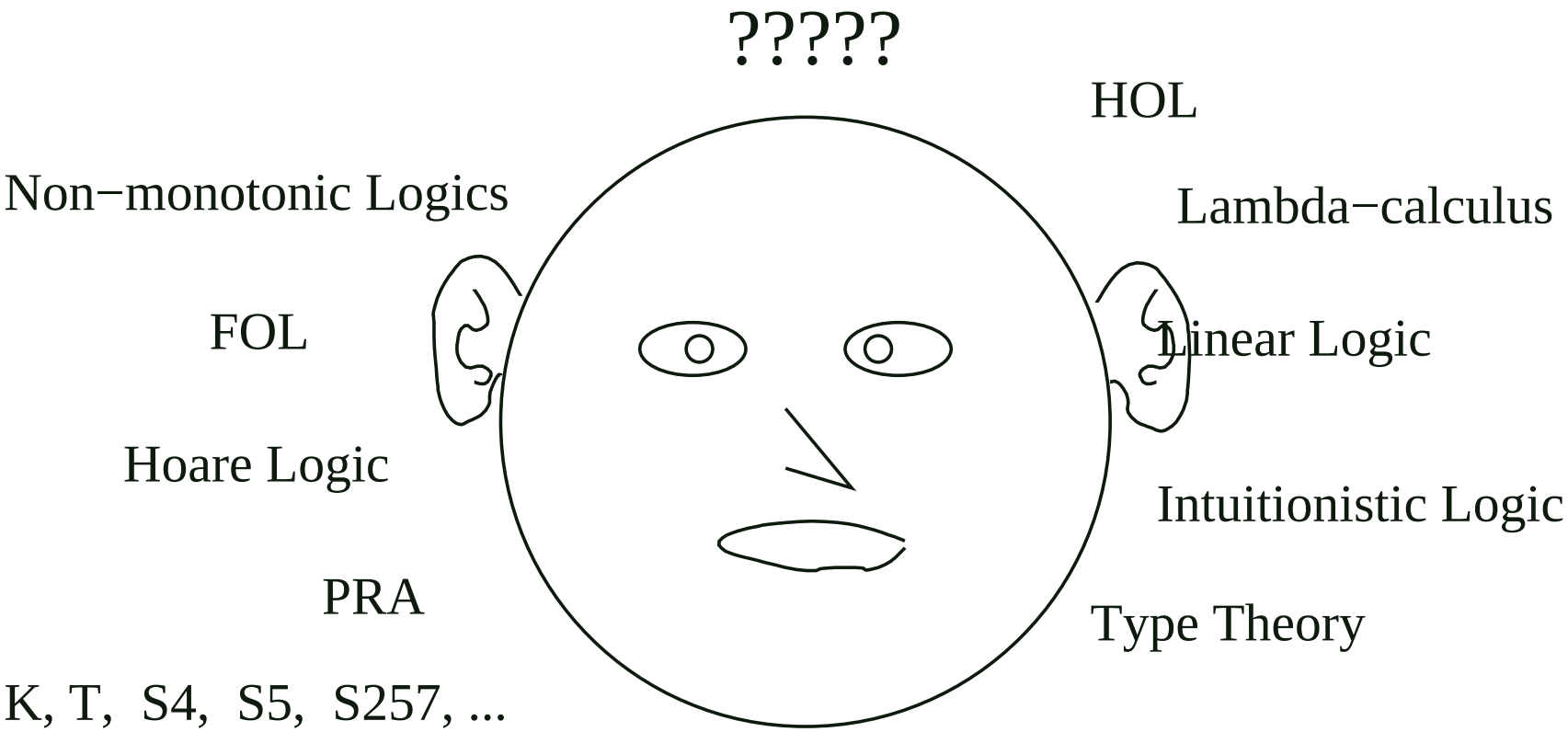
David Basin, Burkhart Wolff, and Jan-Georg Smaus

Overview

- We have studied reasoning in given theories

  Labs used predeveloped `.thy` files.
- How does one encode their own theories? Issues include:
  - Metalogic: formalism for formalizing theories
  - Pragmatics: how to use such a metalogic
- The next two lectures will examine:
  - Representing syntax using simple types
  - Representing proofs using dependent types
- We will be formal

What is the Problem?



Hilbert Presentations, Natural Deduction, Sequent Calculus, ...

Solutions?

- Implement individually

  +/− employment for thousands!

- Embed in a framework logic

  + Implement 'core' only once

  + Shared support for automation

  + Conceptual framework for exploring what a logic is

  +/− Meta-layer between user and logic

  − Makes assumptions about structure of logic

Overview — Encodings in Type Theory

- The λ-Calculus as programming language

$$f(x) = g(x,3) \quad \rightsquigarrow \quad f = \lambda x.\, g\, x\, 3$$

- Simple types classify syntax (o = type of Propositions)

$$\begin{array}{lcll} \bot & \rightsquigarrow & False & \in & o \\ \wedge & \rightsquigarrow & And & \in & o \to o \to o \\ \forall & \rightsquigarrow & All & \in & (i \to o) \to o \end{array}$$

- Dependent types classify rules: pr:$o \to Type$

$$\frac{A \wedge B}{A} \quad \rightsquigarrow \quad andel \in \Pi x : o.\, \Pi y : o.\, pr(and\ x\ y) \to pr(x)$$

Overview (cont.)

- Judgments as Types (syntax in this lecture)

$$\begin{array}{l} \vdots\, P \\ \vdash \phi \end{array} \quad \rightsquigarrow \quad \ulcorner P \urcorner \in pr(\ulcorner \phi \urcorner)$$

  - Models syntax: $\phi \in Prop$ *iff* $\ulcorner \phi \urcorner \in o$
  - Models provability: $\vdash_L \phi$ *iff* $\vdash_{TT} pr(\ulcorner \phi \urcorner)$
  - Models proofs: P *iff* $\ulcorner P \urcorner$
- Correctness of encodings: faithfulness and adequacy

  Requires study of metatheory of metalogic

Simple Type Theory $\lambda^{\rightarrow}$

David Basin, Burkhart Wolff, and Jan-Georg Smaus

The Untyped λ-Calculus

Untyped λ-Calculus

	Conventional		λ-Calculus
Declaration:	$f(x) = g(x, 3)$	$\rightsquigarrow$	$f = \lambda x.\, g\, x\, 3$
Application:	$f(5)$		$(\lambda x.\, g\, x\, 3)(5)$
Reduction:	$g(x, 3)[x \leftarrow 5]$		$(g\, x\, 3)[x \leftarrow 5]$
Result:	$g(5, 3)$		$g\, 5\, 3$

We will use the (typed) λ-calculus to represent the syntax and proofs of deductive systems.

Syntax

$(x \in Var,\ c \in Const)$

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

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

Conventions: iterated λ & left-associated application

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

We may take further syntactic liberties, e.g., $\lambda x.\, x + 5$

Substitution

- Reduction based on substitutions

$$(\lambda x.\, g\, x\, 3)(5) = (g\, x\, 3)[x \leftarrow 5] = g\, 5\, 3$$

- Must respect free and bound variables,

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

- Same problems as with quantifiers

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

Bound, Free, Binding Occurrences

Recall the notions of bound, free, and binding occurrences of variables in a term. Same thing here:

λ-calculus	FOL
$FV(x) :=$	

Bound, Free, Binding Occurrences

Recall the notions of bound, free, and binding occurrences of variables in a term. Same thing here:

λ-calculus	FOL
$FV(x) := \{x\}$	$= FV(x)$
$FV(c) :=$	

Bound, Free, Binding Occurrences

Recall the notions of bound, free, and binding occurrences of variables in a term. Same thing here:

λ-calculus	FOL
$FV(x) := \{x\}$	$= FV(x)$
$FV(c) := \emptyset$	$= FV(c)$
$FV(MN) :=$	

Bound, Free, Binding Occurrences

Recall the notions of bound, free, and binding occurrences of variables in a term. Same thing here:

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

Bound, Free, Binding Occurrences

Recall the notions of bound, free, and binding occurrences of variables in a term. Same thing here:

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

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

Definition of Substitution

$M[x \leftarrow N]$ means substitute N for x in M

1. $x[x \leftarrow N] =$
2. $a[x \leftarrow N] =$
3. $(PQ)[x \leftarrow N] =$
4. $(\lambda x.\, P)[x \leftarrow N] =$
5. $(\lambda y.\, P)[x \leftarrow N] =$

6. $(\lambda y.\, P)[x \leftarrow N] =$

Definition of Substitution

$M[x \leftarrow N]$ means substitute N for x in M

1. $x[x \leftarrow N] = N$
2. $a[x \leftarrow N] = a$ if a is a constant or variable other than x
3. $(PQ)[x \leftarrow N] = (P[x \leftarrow N]Q[x \leftarrow N])$
4. $(\lambda x.\, P)[x \leftarrow N] = \lambda x.\, P$
5. $(\lambda y.\, P)[x \leftarrow N] = \lambda y.\, P[x \leftarrow N]$ if $y \neq x$ and $y \notin FV(N)$
6. $(\lambda y.\, P)[x \leftarrow N] = \lambda z.\, P[y \leftarrow z][x \leftarrow N]$ if $y \neq x$ and $y \in FV(N)$ where z is a variable such that $z \notin FV(NP)$

Definition of Substitution

$M[x \leftarrow N]$ means substitute N for x in M

1. $x[x \leftarrow N] = N$
2. $a[x \leftarrow N] = a$ if a is a constant or variable other than x
3. $(PQ)[x \leftarrow N] = (P[x \leftarrow N]Q[x \leftarrow N])$
4. $(\lambda x.\, P)[x \leftarrow N] = \lambda x.\, P$
5. $(\lambda y.\, P)[x \leftarrow N] = \lambda y.\, P[x \leftarrow N]$ if $y \neq x$ and $y \notin FV(N)$
6. $(\lambda y.\, P)[x \leftarrow N] = \lambda z.\, P[y \leftarrow z][x \leftarrow N]$ if $y \neq x$ and $y \in FV(N)$ where z is a variable such that $z \notin FV(NP)$

Cases similar to those for quantifiers: λ binding is 'generic'.

Substitution: Example

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

Substitution: Example

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

Substitution: Example

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

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

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., $\lambda x.\, xx$.

Reduction: Intuition

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

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

β-reduction replaces a parameter by an argument.

This should propagate into contexts, e.g.

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

Reduction: Definition

- β-reduction: $(\lambda x.M)N \to_\beta M[x \leftarrow N]$
- Rules for contraction (of redices) in contexts:

$$\frac{M \to_\beta M'}{NM \to_\beta NM'} \qquad \frac{M \to_\beta M'}{MN \to_\beta M'N} \qquad \frac{M \to_\beta M'}{\lambda z.M \to_\beta \lambda z.M'}$$

- Reduction is reflexive-transitive closure

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

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

Reduction: Examples

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

Reduction: Examples

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

Reduction: Examples

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

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

Shows Currying

Reduction: Examples

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

So $(\lambda x.\, \lambda y.\, g\, x\, y)a\, b \to_\beta^* g\, a\, b$

Shows Currying

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

Reduction: Examples

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

So $(\lambda x.\, \lambda y.\, g\, x\, y)a\, b \to_\beta^* g\, a\, b$

Shows Currying

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

Shows divergence

Reduction: Examples

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

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

Shows Currying

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

Shows divergence

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

Conversion

- β-conversion: "symmetric closure" of β-reduction

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

Conversion

- β-conversion: "symmetric closure" of β-reduction

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

- α-conversion: bound variable renaming (usually implicitly)

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

Conversion

- β-conversion: "symmetric closure" of β-reduction

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

- α-conversion: bound variable renaming (usually implicitly)

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

- η-conversion: for normal-form analysis

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

reflects an extensional equality on functions.

λ-Calculus Meta-Properties

Confluence (equivalently, Church-Rosser): reduction is order-independent.

For all M, N_1, N_2, if $M \rightarrow_\beta^* N_1$ and $M \rightarrow_\beta^* N_2$, then exists a P where $N_1 \rightarrow_\beta^* P$ and $N_2 \rightarrow_\beta^* P$.

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

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

Uniqueness of Normal Forms

Corollary of the Church-Rosser property:

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

Uniqueness of Normal Forms

Corollary of the Church-Rosser property:

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

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

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

N.B. As a computational formalism, the λ-calculus can represent all computable functions.

The Simply Typed λ-Calculus ($\lambda^{\rightarrow}$)

Simply Typed λ-Calculus — Syntax

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

$$\tau ::= T \mid \tau \rightarrow \tau$$

Simply Typed λ-Calculus — Syntax

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

$$\tau ::= T \mid \tau \rightarrow \tau$$

Examples: $\mathcal{N}$, $\mathcal{N} \rightarrow \mathcal{N}$, $(\mathcal{N} \rightarrow \mathcal{N}) \rightarrow \mathcal{N}$, $\mathcal{N} \rightarrow \mathcal{N} \rightarrow \mathcal{N}$

Simply Typed λ-Calculus — Syntax

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

$$\tau ::= T \mid \tau \rightarrow \tau$$

Examples: $\mathcal{N}$, $\mathcal{N} \rightarrow \mathcal{N}$, $(\mathcal{N} \rightarrow \mathcal{N}) \rightarrow \mathcal{N}$, $\mathcal{N} \rightarrow \mathcal{N} \rightarrow \mathcal{N}$

- Syntax for (raw) terms: λ-calculus augmented with types

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

($x \in Var$, $c \in Const$)

Signatures and Contexts

Generally (in various logic-related formalisms) a signature defines the "fixed" symbols of a language, and a context defines the "variable" symbols of a language.

Signatures and Contexts

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

- a signature Σ is a sequence ($c \in Const$)

$$\Sigma ::= \langle\,\rangle \mid \Sigma, c : \tau$$

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

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

What's the difference to signatures you have seen so far?

Type Assignment Calculus

We now define type judgements: "a term has a type" or "a term is of a type". Generally this depends on a signature Σ and a context Γ. For example

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

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

Type Assignment Calculus

We now define type judgements: "a term has a type" or "a term is of a type". Generally this depends on a signature Σ and a context Γ. For example

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

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

We usually leave Σ implicit and write $\vdash$ instead of $\vdash_{\Sigma}$.

If Γ is empty it is omitted.

Type Assignment Calculus: Rules

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

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

Type Assignment Calculus: Rules

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

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

Note analogy to minimal logic over $\rightarrow$.

β-reduction defined as before, has subject reduction property and is strongly normalizing.

Example 1

$$\vdash \lambda x^{\sigma}.\, \lambda y^{\tau}.\, x :$$

Example 1

$$\vdash \lambda x^{\sigma}.\, \lambda y^{\tau}.\, x : \sigma \rightarrow (\tau \rightarrow \sigma)$$

Example 1

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

Example 1

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

Example 1

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

Example 1

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

Example 1

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

Example 1

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

Note the use of schematic types!

Example 1

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

Note the use of schematic types!

Also note that applications of *hyp* are usually not explicitly marked in proof.

Example 2

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

$$\vdash \lambda f^{\sigma \rightarrow \sigma \rightarrow \tau}.\, \lambda x^{\sigma}.\, f\, x\, x :$$

Example 2

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

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

Example 2

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

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

Example 2

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

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

Example 2

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

$$\dfrac{\dfrac{}{f : \sigma \rightarrow \sigma \rightarrow \tau \vdash \lambda x^{\sigma}.\, f\, x\, x : \sigma \rightarrow \tau}\,\mathit{abs}}{\vdash \lambda f^{\sigma \rightarrow \sigma \rightarrow \tau}.\, \lambda x^{\sigma}.\, f\, x\, x : (\sigma \rightarrow \sigma \rightarrow \tau) \rightarrow \sigma \rightarrow \tau}\,\mathit{abs}$$

Example 2

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

$$\dfrac{\dfrac{\Gamma \vdash f\,x\,x : \tau}{f : \sigma \rightarrow \sigma \rightarrow \tau \vdash \lambda x^{\sigma}.\, f\,x\,x : \sigma \rightarrow \tau}\,\textit{abs}}{\vdash \lambda f^{\sigma \rightarrow \sigma \rightarrow \tau}.\, \lambda x^{\sigma}.\, f\,x\,x : (\sigma \rightarrow \sigma \rightarrow \tau) \rightarrow \sigma \rightarrow \tau}\,\textit{abs}$$

Example 2

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

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

Example 2

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

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

Example 2

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

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma \vdash f\,x : \sigma \rightarrow \tau}\,app \qquad \Gamma \vdash x : \sigma}{\Gamma \vdash f\,x\,x : \tau}\,app}{f : \sigma \rightarrow \sigma \rightarrow \tau \vdash \lambda x^{\sigma}.\, f\,x\,x : \sigma \rightarrow \tau}\,abs}{\vdash \lambda f^{\sigma \rightarrow \sigma \rightarrow \tau}.\, \lambda x^{\sigma}.\, f\,x\,x : (\sigma \rightarrow \sigma \rightarrow \tau) \rightarrow \sigma \rightarrow \tau}\,abs$$

Example 2

$$\Gamma = f : \sigma \rightarrow \sigma \rightarrow \tau, x : \sigma$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\Gamma \vdash f : \sigma \rightarrow \sigma \rightarrow \tau \quad \Gamma \vdash x : \sigma}{\Gamma \vdash f\,x : \sigma \rightarrow \tau}\,app \qquad \Gamma \vdash x : \sigma}{\Gamma \vdash f\,x\,x : \tau}\,app}{f : \sigma \rightarrow \sigma \rightarrow \tau \vdash \lambda x^{\sigma}.\, f\,x\,x : \sigma \rightarrow \tau}\,abs}{\vdash \lambda f^{\sigma \rightarrow \sigma \rightarrow \tau}.\, \lambda x^{\sigma}.\, f\,x\,x : (\sigma \rightarrow \sigma \rightarrow \tau) \rightarrow \sigma \rightarrow \tau}\,abs$$

Example 3

$$\begin{aligned} \Sigma &= f : \sigma \rightarrow \sigma \rightarrow \tau \\ \Gamma &= x : \sigma \end{aligned}$$

$$\Gamma \vdash f\, x\, x : \tau$$

Example 3

$$\begin{aligned}\Sigma &= f : \sigma \rightarrow \sigma \rightarrow \tau \\ \Gamma &= x : \sigma\end{aligned}$$

$$\dfrac{\dfrac{\dfrac{f : \sigma \rightarrow \sigma \rightarrow \tau \in \Sigma}{\Gamma \vdash f : \sigma \rightarrow \sigma \rightarrow \tau}\,\textit{assum} \qquad \Gamma \vdash x : \sigma}{\Gamma \vdash f\,x : \sigma \rightarrow \tau}\,\textit{app} \qquad \Gamma \vdash x : \sigma}{\Gamma \vdash f\,x\,x : \tau}\,\textit{app}$$

Note that this time, f is a constant.

Example 3

$$\begin{aligned}\Sigma &= f : \sigma \rightarrow \sigma \rightarrow \tau \\ \Gamma &= x : \sigma\end{aligned}$$

$$\dfrac{\dfrac{\Gamma \vdash f : \sigma \rightarrow \sigma \rightarrow \tau \qquad \Gamma \vdash x : \sigma}{\Gamma \vdash f\,x : \sigma \rightarrow \tau}\,app \qquad \Gamma \vdash x : \sigma}{\Gamma \vdash f\,x\,x : \tau}\,app$$

Note that this time, f is a constant.

We will often suppress applications of *assum*.

Example 4: First-Order Syntax within $\lambda^{\rightarrow}$

- Propositional logic

$$P ::= x \mid \neg P \mid P \wedge P \mid P \rightarrow P \ldots$$

- Programming languages/algebraic specification

datatype *Prop* = *VarInject* of *Variable* | *not* of *Prop*
| *and* of *Prop**Prop | *imp* of *Prop**Prop

- $\lambda^{\rightarrow}$ approach
  - Type declarations for context $\mathcal{B} = \{o\}$
  - Signature types constants:

$$\Sigma = \{not : o \rightarrow o, and : o \rightarrow o \rightarrow o, imp : o \rightarrow o \rightarrow o\}$$

- Context types propositional variables

Example 4: First-Order Syntax within $\lambda^{\rightarrow}$(cont.)

- Example: $a : o \vdash imp(not\, a)a : o$

$$\dfrac{\dfrac{a : o \vdash imp : o \rightarrow o \qquad \dfrac{a : o \vdash not : o \rightarrow o \quad a : o \vdash a : o}{a : o \vdash not\, a : o}}{a : o \vdash imp(not\, a) : o \rightarrow o} \qquad a : o \vdash a}{a : o \vdash imp(not\, a)a : o}$$

- Non example: $a : o \vdash not(imp\, a)a : o$

$$\dfrac{a : o \vdash not : o \rightarrow o \qquad \dfrac{a : o \vdash imp : o \rightarrow o \rightarrow o \quad a : o, \vdash a : o}{a : o \vdash imp\, a : o \rightarrow o}}{???}$$

No proof possible! (requires analysis of normal forms)

Example 5: Encoding Binding in FOL

- This also works for languages with quantifiers !

$$\begin{array}{lll} \textit{Terms } T & ::= & x \mid 0 \mid sT \mid T+T \mid T \times T \\ \textit{Formulae } F & ::= & T = T \mid \neg F \mid F \wedge F \mid \ldots \\ & & \forall x.\, F \mid \exists x.\, F \end{array}$$

- Type declarations for context $\mathcal{B} = \{i, o\}$
- Signature $\Sigma = \Sigma_T \cup \Sigma_P \cup \Sigma_Q$:

$$\begin{array}{lll} \Sigma_T & = & \{0 : i,\ s : i \rightarrow i,\ plus : i \rightarrow i \rightarrow i,\ times : i \rightarrow i \rightarrow i\} \\ \Sigma_P & = & \{eq : i \rightarrow i \rightarrow o,\ not : o \rightarrow o,\ and : o \rightarrow o \rightarrow o,\ \ldots\} \\ \Sigma_Q & = & \{all : (i \rightarrow o) \rightarrow o,\ exists : (i \rightarrow o) \rightarrow o\} \end{array}$$

Example 5: Encoding Binding in FOL (cont)

- Faithfulness/adequacy: terms and formulae represented by (canonical) members of i and o

$$
\begin{array}{lcl}
0 + s0 & \Leftrightarrow & plus\,0\,(s0) \\
\forall x.\, x = x & \Leftrightarrow & all(\lambda x^i.\, eq\, x\, x) \\
\forall x.\, \exists y.\, \neg(x + x = y) & \Leftrightarrow & all(\lambda x^i.\, exists(\lambda y^i.\, not\,(eq\,(plus\, x
\end{array}
$$

- Example derivation

$$
\dfrac{\vdash all : (i \rightarrow o) \rightarrow o \qquad \dfrac{\dfrac{\dfrac{x : i \vdash eq : i \rightarrow i \rightarrow o \qquad x : i \vdash x : i}{x : i \vdash eq\, x : i \rightarrow o} \qquad x : i \vdash x :}{x : i \vdash eq\, x\, x : o}}{\vdash \lambda x^i.\, eq\, x\, x : i \rightarrow o}}{\vdash all(\lambda x^i.\, eq\, x\, x) : o}
$$

More Detailed Explanations

$3 + 5 = 8$?

As you might guess, the formalism of the λ-calculus is not directly related to usual arithmetic and so it is not built into this formalism that $3 + 5$ should evaluate to 8. However, it may be a reasonable choice, depending on the context, to extend the λ-calculus is this way, but this is not our concern at the moment.

Var and $Const$

Similarly as for first-order logic, a language of the untyped λ-calculus is characterized by giving a set of variables and a set of constants.

One can think of $Const$ as a signature.

Note that $Const$ could be empty.

Note also that the word constant has a different meaning in the λ-calculus from that of first-order logic. In both formalisms, constants are just symbols.

In first-order logic, a constant is a special case of a function symbol, namely a function symbol of arity 0.

In the λ-calculus, one does not speak of function symbols. In the untyped λ-calculus, any λ-term (including a constant) can be applied to another term, and so any λ-term can be called a "unary function". A constant being applied to a term is something which would contradict

the intuition about constants in first-order logic. So for the λ-calculus, think of constant as opposed to a variable, an application, or an abstraction.

How do We Call those Terms?

A λ-term can either be

- a variable (case x), or
- a constant (case c), or
- an application of a λ-term to another λ-term (case (ee)), or
- an abstraction over a variable x (case $(\lambda x.\, e)$).

Backus-Naur Form

A notation like

$$
\begin{aligned}
e &::= x \mid c \mid (ee) \mid (\lambda x.\, e) \\
\tau &::= T \mid \tau \to \tau \\
e &::= x \mid c \mid (ee) \mid (\lambda x^{\tau}.\, e) \\
P &::= x \mid \neg P \mid P \wedge P \mid P \to P \ldots
\end{aligned}
$$

for specifying syntax is called Backus-Naur form (BNF) for expressing grammars. For example, the first BNF-clause reads: a λ-term can be
a variable, or
a constant, or
a λ-term applied to a λ-term, or
a λ-abstraction, which is a λ-term of the form $\lambda x.\, e$, where e is a λ-term.

The BNF is a very common formalism for specifying syntax, e.g., of

programming languages. See here or here.

(λ-)Terms

So just like first-order logic, the λ-calculus has a syntactic category called terms. Bit the word "term" has a different meaning for the λ-calculus than for first-order logic, and so one can say λ-term for emphasis.

Note that at this stage, we have no syntactic category called "formula" for the λ-calculus.

λ-Calculus: Notational Conventions

We write $\lambda x_1 x_2 \ldots x_n.e$ instead of $\lambda x_1.(\lambda x_2.(\ldots e)\ldots)$.

$e_1\ e_2 \ldots e_n$ is equivalent to $(\ldots(e_1\ e_2)\ldots e_n)\ldots$, not $(e_1(e_2 \ldots e_n)\ldots)$.

Note that this is in contrast to the associativity of logical operators.

There are some good reasons for these conventions.

Infix Notation

Strictly speaking, $\lambda x.\, x + 5$ does not adhere to the definition of syntax of λ-terms, at least if we parse it in the usual way: $+$ is an infix constant applied to arguments x and 5.

If we parse $x + 5$ as $((x+)5)$, i.e., x applied to (the constant) $+$, and the resulting term applied to (the constant) 5, then $\lambda x.\, x + 5$ would indeed adhere to the definition of syntax of λ-terms, but of course, this is pathological and not intended here.

It is convenient to allow for extensions of the syntax of λ-terms, allowing for:

- application to several arguments rather than just one;
- infix notation.

Such an extension is inessential for the expressive power of the

λ-calculus. Instead of having a binary infix constant $+$ and writing $\lambda x.\, x + 5$, we could have a constant $plus$ according to the original syntax and write $\lambda x.\, ((plus\, x)\, 5)$ (i.e., write $+$ in a Curryed way).

Reduction

Reduction is the notion of "computing", or "evaluation", in the λ-calculus.

Notations for Substitutions

Here we use the notation $e[x \leftarrow t]$ for the term obtained from e by replacing x with t. There is also the notation $e[t/x]$, and confusingly, also $e[x/t]$. We will attempt to be consistent within this course, but be aware that you may find such different notations in the literature.

λ Binding Is 'Generic'

Recall the definition of substitution for first-order logic.

We observe that binding and substitution are some very general concepts. So far, we have seen four binding operators: $\exists$, $\forall$ and λ, and set comprehensions. The λ operator is the most generic of those operators, in that it does not have a fixed meaning hard-wired into it in the way that the quantifiers do. In fact, it is possible to have it as the only operator on the level of the metalogic. We will see this later.

Avoiding Capture

If it wasn't for clause 6, i.e., if we applied clause 5 ignoring the requirement on freeness, then $(\lambda x.\, xy)[y \leftarrow x]$ would be $\lambda x.\, xx$.

Parameters and Arguments

In the λ-term $(\lambda x.M)N$, we say that N is an argument (and the function $\lambda x.M$ is applied to this argument), and every occurrence of x in M is a parameter (we say this because x is bound by the λ).

This terminology may be familiar to you if you have experience in functional programming, but actually, it is also used in the context of function and procedure declarations in imperative programming.

Propagation into Contexts

In

$$\lambda x.(\underline{(\lambda x.\, x + 5)(3)}),$$

the underlined part is a subterm occurring in a context. β-reduction should be applicable to this subterm.

Like a Proof System

As you see, β-reduction is defined using rules (two of them being axioms, the rest proper rules) in the same way that we have defined proof systems for logic before. Note that we wrote the first axiom defining β-reduction without a horizontal bar.

Redex

In a λ-term, a subterm of the form $(\lambda x.\, M)N$ is called a redex (plural redices). It is a subterm to which β-reduction can be applied.

Currying

You may be familiar with functions taking several arguments, or equivalently, a tuple of arguments, rather than just one argument.

In the λ-calculus, but also in functional programming, it is common not to have tuples and instead use a technique called Currying (Schönfinkeln in German). So instead of writing $g(a, b)$, we write $g\,a\,b$, which is read as follows: g is a function which takes an argument a and returns a function which then takes an argument b.

Recall that application associates to the left, so $g\,a\,b$ is read $(g\,a)\,b$.

Currying will become even clearer once we introduce the typed λ-calculus.

Divergence

We say that a β-reduction sequence diverges if it is infinite.

Note that for $(\lambda xy.\, y)((\lambda x.\, xx)(\lambda x.\, xx))$, there is a finite β-reduction sequence

$$(\lambda xy.\, y)((\lambda x.\, xx)(\lambda x.\, xx)) \to_\beta \lambda y.\, y$$

but there is also a diverging sequence

$$(\lambda xy.\, y)((\lambda x.\, xx)(\lambda x.\, xx)) \to_\beta (\lambda xy.\, y)((\lambda x.\, xx)(\lambda x.\, xx)) \to_\beta \ldots$$

α-Conversion

α-conversion is usually applied implicitly, i.e., without making it an explicit step. So for example, one would simply write:

$$\lambda z.\, z =_\beta (\lambda x.\, xx)\lambda r.\, r$$

η-Conversion

η-conversion is defined as

$$M =_\eta \lambda x.\,(M x) \quad \textit{if}\; x \notin FV(M)$$

It is needed for reasoning about normal forms.

$$g\,x =_\eta \lambda y.\,g\,x\,y \quad \text{reflects} \quad g\,x\,b =_\beta (\lambda y.\,g\,x\,y)b$$

More specifically: if we did not have the η-conversion rule, then $g\,x$ and $\lambda y.\,g\,x\,y$ would not be "equivalent" up to conversion. But that seems unreasonable, because they behave the same way when applied to b. Applied to b, both terms can be converted to $g\,x\,b$. This is why it is reasonable to introduce a rule such that $g\,x$ and $\lambda y.\,g\,x\,y$ are "equivalent" up to conversion.

Confluence and Church-Rosser

A reduction $\rightarrow$ is called confluent if

for all M, N_1, N_2, if $M \rightarrow^* N_1$ and $M \rightarrow^* N_2$, then there exists a P where $N_1 \rightarrow^* P$ and $N_2 \rightarrow^* P$.

A reduction is called Church-Rosser if

for all N_1, N_2, if $N_1 \overset{*}{\leftrightarrow} N_2$, 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

$$M_1 \to M_2 \to M_3 \to M_4 \leftarrow M_5 \leftarrow M_6 \to M_7 \leftarrow M_8 \leftarrow M_9$$

then we would write $M_1 \overset{*}{\leftrightarrow} M_9$.

Confluence is equivalent to the Church-Rosser property [BN98, page 10].

λ-Calculus Metaproperties

By metaproperties, we mean properties about reduction and conversion sequences in general.

Turing Completeness

The untyped λ-calculus is Turing complete. This is usually shown not by mimicking a Turing machine in the λ-calculus, but rather by exploiting the fact that the Turing computable functions are the same class as the μ-recursive functions. In a lecture on theory of computation, you have probably learned that the μ-recursive functions are obtained from the primitive recursive functions by so-called unbounded minimalization, while the primitive recursive functions are built from the 0-place zero function, projection functions and the successor function using composition and primitive recursion [LP81].

The proof that the untyped λ-calculus can compute all μ-recursive functions is thus based on showing that each of the mentioned ingredients can be encoded in the untyped λ-calculus. While we are not going to study this, one crucial point is that it should be possible to

encode the natural numbers and the arithmetic operations in the untyped λ-calculus.

Term Language

We also say that we have defined a term language. A particular language is given by a signature, although for the untyped λ-calculus this is simply the set of constants $Const$.

Type Language

We can say that we define a type language, i.e., a language consisting of types. A particular type language is characterized by a giving a set of base types $\mathcal{B}$. One might also call $\mathcal{B}$ a type signature.

A typical example of a set of base types would be $\{\mathcal{N}, bool\}$, where $\mathcal{N}$ represents the natural numbers and $bool$ the Boolean values $\bot$ and $\top$.

All that matters is that $\mathcal{B}$ is some fixed set "defined by the user".

Types: Intuition

The type $\mathcal{N} \to \mathcal{N}$ is the type of a function that takes a natural number and returns a natural number.

The type $(\mathcal{N} \to \mathcal{N}) \to \mathcal{N}$ is the type of a function that takes a function, which takes a natural number and returns a natural number, and returns a natural number.

Types Are Right-Associative

To save parentheses, we use the following convention: types associate to the right, so $\mathcal{N} \to \mathcal{N} \to \mathcal{N}$ stands for $\mathcal{N} \to (\mathcal{N} \to \mathcal{N})$.

Recall that application associates to the left. This may seem confusing at first, but actually, it turns out that the two conventions concerning associativity fit together very neatly.

Raw Terms

In the context of typed versions of the λ-calculus, raw terms are terms built ignoring any typing conditions. So raw terms are simply terms as defined for the untyped λ-calculus, possibly augmented with type superscripts.

Augmenting with Types

So far, this is just syntax!

The notation $(\lambda x^{\tau}.\, e)$ simply specifies that binding occurrences of variables in simple type theory are tagged with a superscript, where the use of the letter τ makes it clear (in this particular context) that the superscript must be some type, defined by the grammar we just gave.

Var and $Const$

Var and $Const$ are the sets of variables and constants, respectively, as for the untyped λ-calculus.

Sequences

A sequence is a collection of objects which differs from sets in that a sequence contains the objects in a certain order, and there can be multiple occurrences of an object.

We write a sequence containing the objects $o_1, \ldots, o_n$ 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 $\langle o, o_1, \ldots, o_n \rangle$.

A empty sequence is denoted by $\langle\, \rangle$.

Type Binding

We call an expression of the form $x : \tau$ or $c : \tau$ a type binding.

The use of the letter τ makes it clear (in this particular context) that the superscript must be some type, defined by the grammar we just gave.

Signatures in Various Formalisms

For propositional logic, we did not use the notion of signature, although we mentioned that strictly speaking, there is not just the language of propositional logic, but rather a language of propositional logic which depends on the choice of the variables.

In first-order logic, a signature was a pair $(\mathcal{F}, \mathcal{P})$ defining the function and predicate symbols, although strictly speaking, the signature should also specify the arities of the symbols in some way. Recall that we did not bother to fix a precise technical way of specifying those arities. We were content with saying that they are specified in "some unambiguous way".

In sorted logic, the signature must also specify the sorts of all symbols. But we did not study sorted logic in any detail.

In the untyped λ-calculus, the signature is simply the set of constants.

Summarizing, we have not been very precise about the notion of a

signature so far, since technically speaking, it was not strictly necessary to have this notion.

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 $c \in Const$, and in fact, that Σ is clear from the context as well. Since Σ contains an expression of the form $c : \tau$ for each $c \in Const$, it is redundant to give $Const$ explicitly. It is sufficient to give Σ.

Type Judgement

The expression

$$\Gamma \vdash_{\Sigma} c\, x : \sigma$$

is called a type judgement. It says that given the signature $\Sigma = x : \tau$ and the context $\Gamma = c : \tau \to \sigma$, the term

$c\, x$ has type σ or

$c\, x$ is of type σ or

$c\, x$ is assigned type σ.

Recall that you have seen other judgements before.

$\in$ for Sequences?

Recall that Σ is a sequence. By abuse of notation, we sometimes identify this sequence with a set and allow ourselves to write $c : \tau \in \Sigma$.

We may also write $\Sigma \subseteq \Sigma'$ meaning that $c : \tau \in \Sigma$ implies $c : \tau \in \Sigma'$.

System of Rules

Type assignment is defined as a system of rules for deriving type judgements, in the same way that we have defined derivability judgements for logics, and β-reduction for the untyped λ-calculus.

Minimal Logic over $\rightarrow$

Recall the sequent rules of the $\rightarrow/\wedge$ fragment of propositional logic. Consider now only the $\rightarrow$ fragment. We call this fragment minimal logic over $\rightarrow$.

If you take the rule

$$\Gamma, x : \tau, \Delta \vdash x : \tau \quad \textit{hyp}$$

of $\lambda^{\rightarrow}$ and throw away the terms (so you keep only the types), you obtain essentially the rule for assumptions

$$\Gamma \vdash A \quad (\text{where } A \in \Gamma)$$

of propositional logic.

Likewise, if you do the same with the rule

$$\frac{\Gamma \vdash e : \sigma \to \tau \quad \Gamma \vdash e' : \sigma}{\Gamma \vdash ee' : \tau}\ \textit{app}$$

of $\lambda^{\to}$, you obtain essentially the rule

$$\frac{\Gamma \vdash A \to B \quad \Gamma \vdash A}{\Gamma \vdash B}\ \textit{\to-E}$$

of propositional logic.

Finally, if you do the same with the rule

$$\frac{\Gamma, x : \sigma \vdash e : \tau}{\Gamma \vdash \lambda x^{\sigma}.\, e\ : \sigma \to \tau}\ \textit{abs}$$

of $\lambda^{\rightarrow}$, you obtain essentially the rule

$$\frac{A, \Gamma \vdash B}{\Gamma \vdash A \rightarrow B} \rightarrow\text{-}I$$

of propositional logic.

Note that in this setting, there is no analogous propositional logic rule for

$$\frac{c : \tau \in \Sigma}{\Gamma \vdash c : \tau} \textit{assum}$$

So for the moment, we can observe a close analogy between $\lambda^{\rightarrow}$, for Σ being empty, and the $\rightarrow$ fragment of propositional logic, which is also called minimal logic over $\rightarrow$.

Such an analogy between a type theory (of which $\lambda^{\rightarrow}$ is an example) and a logic is referred to in the literature as Curry-Howard isomorphism. One

also speaks of propositions as types [GLT89]. The isomorphism is so fundamental that it is common to characterize type theories by the logic they represent, so for example, one might say:

$\lambda^{\rightarrow}$ is the type theory of minimal logic over $\rightarrow$.

Subject Reduction

Subject reduction is the following property: reduction does not change the type of a term, so if $\vdash_\Sigma M : \tau$ and $M \to_\beta N$, then $\vdash_\Sigma N : \tau$.

(Strongly) Normalizing β-Reduction

The simply-typed λ-calculus, unlike the untyped λ-calculus, is normalizing, that is to say, every term has a normal form. Even more, it is strongly normalizing, that is, this normal form is reached regardless of the reduction order.

An Alternative for *hyp*

One could also formulate *hyp* as follows:

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}\ \textit{hyp}$$

That would be in close analogy to LF, a system not tretaed here.

Schematic Types

In this example, you may regard σ and τ as base types (this would require that $\sigma, \tau \in \mathcal{B}$), but in fact, it is more natural to regard them as metavariables standing for arbitrary types. Whatever types you substitute for σ and τ, you obtain a derivation of a type judgement.

This is in analogy to schematic derivations in a logic.

Note also that Σ is irrelevant for the example and hence arbitrary.

Constants vs. Variables

In Example 3, we have $f : \sigma \to \sigma \to \tau \in \Sigma$, and so f is a constant.

In Example 2, we have $f : \sigma \to \sigma \to \tau \in \Gamma$, and so f is a variable.

Looking at the different derivations of the type judgement $\Gamma \vdash f\,x\,x : \tau$ in Examples 2 and 3, you may find that they are very similar, and you may wonder: What is the point? Why do we distinguish between constants and variables?

In fact, one could simulate constants by variables. When setting up a type theory or programming language, there are choices to be made about whether there should be a distinction between variables and constants, and what it should look like. There is a famous epigram by Alan Perlis:

> One man's constant is another man's variable.

For our purposes, it is much clearer conceptually to make the distinction. For example, if we want to introduce the natural numbers in our $\lambda^{\rightarrow}$ language, then it is natural that there should be constants $1, 2, \ldots$ denoting the numbers. If $1, 2, \ldots$ were variables, then we could write strange expressions like $\lambda 2^{\mathcal{N} \rightarrow \mathcal{N}}. y$, so we could use 2 as a variable of type $\mathcal{N} \rightarrow \mathcal{N}$.

(Parametric) Polymorphism

In functional programming, you will come across functions that operate uniformly on many different types. For example, a function $append$ for concatenating two lists works the same way on integer lists and on character lists. Such functions are called polymorphic.

More precisely, this kind of polymorphism, where a function does exactly the same thing regardless of the type instance, is called parametric polymorphism, as opposed to ad-hoc polymorphism.

In a type system with polymorphism, the notion of base type (which is just a type constant, i.e., one symbol) is generalized to a type constructor with an arity ≥ 0. A type constructor of arity n applied to n types is then a type. For example, there might be a type constructor $list$ of arity 1, and int of arity 0. Then, $int\ list$ is a type.

Note that application of a type constructor to a type is written in postfix

notation, unlike any notation for function application we have seen. However, other conventions exist, even within Isabelle.

See [Pau96, Tho95b, Tho99] for details on the polymorphic type systems of functional programming languages.

Ad-hoc Polymorphism

Ad-hoc polymorphism, also called overloading, refers to functions that do different (although usually similar) things on different types. For example, a function $\leq$ may be defined as $'a' \leq' b' \ldots$ 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, but also somewhat different from type classes

Type classes are a way of "making ad-hoc polymorphism less ad-hoc" [HHPW96, WB89].

Type Classes

Type classes are a way of "making ad-hoc polymorphism less ad-hoc" [HHPW96, WB89].

Type classes are used to group together types with certain properties, in particular, types for which certain symbols are defined.

For example, for some types, a symbol $\leq$ (which is a binary infix predicate) may exist and for some not, and we could have a type class ord containing all types for which it exists.

Suppose you want to sort a list of elements (smaller elements should come before bigger elements). This is only defined for elements of a type for which the symbol $\leq$ exists.

Note that while a symbol such as $\leq$ may have a similar meaning for different types (for example, integers and reals), one cannot say that it means exactly the same thing regardless of the type of the argument to

which it is applied. In fact, $\leq$ has to be defined separately for each type in ord.

This is in contrast to parametric poymorphism, but also somewhat different from ad-hoc polymorphism: The types of the symbols must not be declared separately. E.g., one has to declare only once that $\leq$ is of type $(a :: ord, \alpha)$.

Polymorphic Type Language

As before, we define a type language, i.e., a language consisting of types, and a particular type language is characterized by a giving a certain set of symbols $\mathcal{B}$. But unlike before, $\mathcal{B}$ is now a set of type constructors. Each type constructor has an arity associated with it just like a function in first-order logic. The intention is that a type constructor may be applied to types.

Following the conventions of ML [Pau96], we write types in postfix notation, something we have not seen before. I.e., the type constructor comes after the arguments it is applied to.

It makes perfect sense to view the function construction arrow $\rightarrow$ as type constructor, however written infix rather than postfix.

So the $\mathcal{B}$ is some fixed set "defined by the user", but it should definitely always include $\rightarrow$.

Type Substitutions

A type substitution replaces a type variable by a type, just like in first-order logic, a substitution replaces a variable by a term.

Syntactic Classes

A syntactic class is a class of types for which certain symbols are declared to exist. Isabelle has a syntax for such declarations. E.g., the declaration

```
sort ord < term
const <= : ['a::ord, 'a] => bool
```

may form part of an Isabelle theory file. It declares a type class ord which is 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 $term$. 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 in Isabelle. Note that the second occurrence of α is written without $::\ ord$. This is because it is enough to state the class constraint once.

Note also that $[\alpha :: ord, \alpha] => bool$ is in fact just another way of writing $\alpha :: ord => \alpha => bool$, similarly as for goals.

Haskell [HHPW96] has type classes but ML [Pau96] hasn't.

Axiomatic Classes

In addition to declaring the syntax of a type class, one can axiomatize the semantics of the symbols. Again, Isabelle has a syntax for such declarations. E.g., the declaration

```
axclass order < ord
  order_refl: ''x <= x ''
  order_trans: ''[| x <= y; y <= z |] ==> x <= z''
  ...
```

may form part of an Isabelle theory file. It declares an axiomatic type class $order$ which is a subclass of ord defined above.

The next two lines are the axioms. Here, `order_refl` and `order_trans` are the names of the axioms. Recall that $\Longrightarrow$ is the implication symbol in Isabelle (that is to say, the metalevel implication).

Whenever an Isabelle theory declares that a type is a member of such a class, it must prove those axioms.

The rationale of having axiomatic classes is that it allows for proofs that hold in different but similar mathematical structures to be done only once. So for example, all theorems that hold for dense orders can be proven for all dense orders with one single proof.

Renaming

Whenever a rule is applied, the metavariables occurring in it must be renamed to fresh variables to ensure that no metavariable in the rule has been used in the proof before.

The notion fresh is often casually used in logic, and it means: this variable has never been used before. To be more precise, one should say: never been used before in the relevant context.

Unification

The mechanism to instantiate metavariables as needed is called (higher-order) unification. Unification is the process of finding a substitution that makes two terms equal.

We will later see more formally what it is and also where it is used.

Type Class Syntax

$$\kappa \quad ::= \quad ord \mid order \mid lattice \mid \ldots$$

is a grammar defining what type classes are (syntactically). κ is the non-terminal we use for "type class". However, the grammar given here is incomplete (there are "...") 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 $lattice$.

The grammar does not tell us what syntax is used to declare the type classes. We have seen an example of that previously.

Type Constructor Syntax

$$\chi \quad ::= \quad bool \mid _ \rightarrow _ \mid ind \mid _\ list \mid _\ set \ldots$$

is a grammar defining what type constructors are (syntactically). χ is the non-terminal we use for "type constructor". As before, the grammar given here is incomplete (there are "...") and just exemplary.

Note also that an _ is used to denote the arity of a type constructor.

- _ $list$ means that $list$ is unary type constructor;
- _ $\rightarrow$ _ means that $\rightarrow$ is a binary infix type constructor.

The notation using _ is slightly abusive since the _ is not actually part of the type constructor (and the grammar is supposed to define type constructors). _ $list$ is not a type constructor; $list$ is a type constructor.

So the set of type constructors involved in an Isabelle theory is a finite set of names (written lower-case) with each having an arity associated,

typically including $bool$, $\rightarrow$, and $list$. Note however that $bool$ is fundamental (since object level predicates are modeled functions taking terms to a Boolean), and so is $\rightarrow$, the constructor of the function space between two types.

The grammar does not tell us what syntax is used to declare the type constructors.

$\rightarrow$ as Type Constructor

In $\lambda^{\rightarrow}$, types were built from base types using a "special symbol" $\rightarrow$. 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.

Polymorphic Types Syntax

$$\tau \quad ::= \quad \alpha \mid \alpha :: \kappa \mid (\tau, \ldots, \tau)\,\chi \qquad (\alpha \text{ is type variable})$$

is a grammar defining what polymorphic types are (syntactically). As before, τ is the non-terminal we use for (now: polymorphic) types.

This grammar is not exemplary but generic, and it deserves a closer look.

A type variable is a variable that stands for a type, as opposed to a term. We have not given a grammar for type variables, but assume that there is a countable set of type variables disjoint from the set of term variables. We use α as the non-terminal for a type variable (abusing notation, we often also use α to denote an actual type variable).

First, note that a type variable may be followed by a class constraint $:: \kappa$ (recall that κ is the non-terminal for type classes). However, a type variable is not necessarily followed by such a constraint, for example if

the type variable already occurs elsewhere and is constrained in that place. We have already seen this.

Moreover, a polymorphic type is obtained by preceding a type constructor with a tuple of types. The arity of the tuple must be equal to the declared arity of the type constructor.

It is not shown here that for some special type constructors, such as $\rightarrow$, the argument may also be written infix.

Type Instantiation

The assumption and hypothesis rules have an assumption of the form $\tau \prec \sigma$.

The symbol $\prec$ is an ordering on types, induced by the subclass ordering $\prec$ on type classes. $\tau \prec \sigma$ means that τ is an instance of σ, and τ is in a type class c, and σ is in a type class d, such that $c \prec d$.

One can also write $\tau :: c$ and $\sigma :: d$. We have previously seen the notation $\alpha :: c$ for a type variable constrained to be in class c. We regarded the whole expression $\alpha :: c$ as a type, but we have also seen that a type variable is not necessarily followed by such a constraint.

Here, τ, σ are arbitrary types, not necessarily a type variables. For a type τ other than a type variable, the expression $\tau :: c$ must be read as an assertion that τ is in type class c.

One can formalize precisely when the type class declarations of an

Isabelle theory entail the assertion that a type is in a certain type class, but we do not go into these details here.

Note that $\prec$ is reflexive.

Consult [HHPW96, Nip93] for details on type classes.

Type Construction

Type construction is the problem of given a Σ, Γ and e, finding a t such that $\Sigma\,,\,\Gamma \vdash e : \tau$.

Sometimes one also considers the problem where Γ is unknown and must also be constructed.

Term Congruence

$\alpha\beta\eta$-conversion is defined as for $\lambda^{\rightarrow}$. Given two (extended) λ-terms e and e', it is decidable whether $e =_{\alpha\beta\eta} e'$.

Solutions for Unification Problems

A solution for $?X + ?Y =_{\alpha\beta\eta} x + x$ is $[x/?X, x/?Y]$.

A solution for $?P(x) =_{\alpha\beta\eta} x + x$ is $[(\lambda y.y + y)/?P]$.

A solution for $f(?X x) =_{\alpha\beta\eta} ?Y\, x$ is $[(\lambda z.z)/?X, (\lambda z.f\, z)/?Y]$.

Unification Modulo

Unification of terms e, e' modulo $\alpha\beta$ means finding a substitution θ for metavariables such that $\theta(e) =_{\alpha\beta} \theta(e')$.

Likewise, unification of terms e, e' modulo $\alpha\beta\gamma$ means finding a substitution σ for metavariables such that $\sigma(e) =_{\alpha\beta\gamma} \sigma(e')$.

Metatheory I: Syntax

David Basin

ETH Zurich
8.11.04

Overview

- We have studied reasoning in given theories

  Labs used predeveloped .thy files.

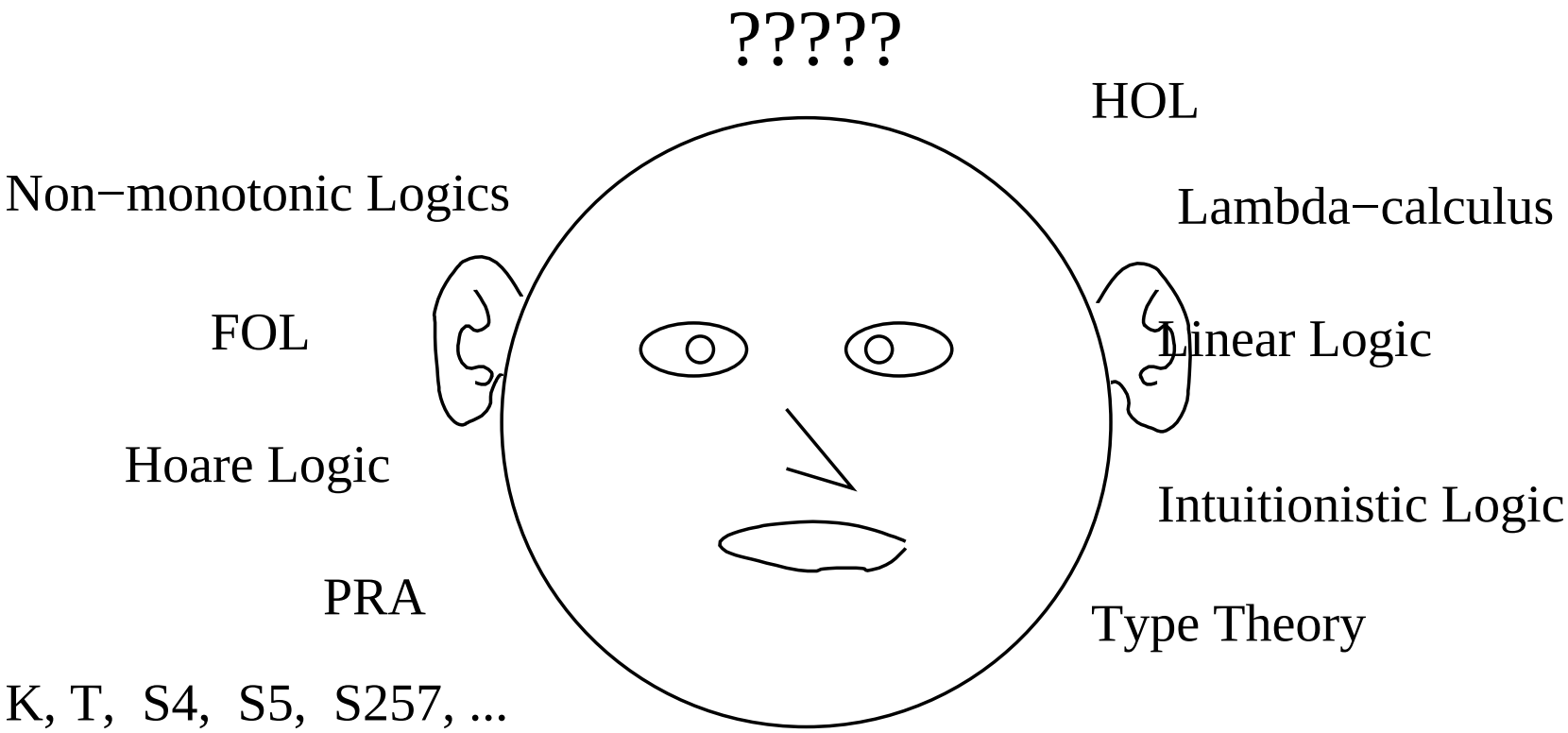
- How does one encode their own theories? Issues include:
  - Metalogic: formalism for formalizing theories
  - Pragmatics: how to use such a metalogic

- The next two lectures will examine:
  - Representing syntax using simple types
  - Representing proofs using dependent types

- We will be formal

  Labs will provide practical experience using formal metatheories

What is the Problem?



Hilbert Presentations, Natural Deduction, Sequent Calculus, ...

Solutions?

- Implement individually

  +/− employment for thousands !

- Embed in a framework logic

  + Implement ‘core’ only once
  + Shared support for automation
  + Conceptual framework for exploring what a logic is
  +/− Meta-layer between user and logic
  − Makes assumptions about structure of logic

Overview — Syntactic Encodings in Type Theory

- The λ-Calculus as programming language

$$f(x) = g(x,3) \quad \rightsquigarrow \quad f = \lambda x.\, g\, x\, 3$$

- Simple types classify syntax (o = type of Propositions)

$$\begin{array}{lcll} \bot & \rightsquigarrow & False & \in \; o \\ \wedge & \rightsquigarrow & And & \in \; o \to o \to o \\ \forall & \rightsquigarrow & All & \in \; (i \to o) \to o \end{array}$$

- Dependent types classify rules: pr:$o \to Type$

$$\frac{A \wedge B}{A} \quad \rightsquigarrow \quad andel \in \Pi x : o.\, \Pi y : o.\, pr(and\; x\; y) \to pr(x)$$

Overview (cont.)

- Judgments as Types (syntax in this lecture)

$$\begin{array}{l} \vdots\, P \\ \vdash \phi \end{array} \quad \rightsquigarrow \quad \ulcorner P \urcorner \in pr(\ulcorner \phi \urcorner)$$

  - Models syntax: $\phi \in Prop$ *iff* $\ulcorner \phi \urcorner \in o$
  - Models provability: $\vdash_L \phi$ *iff* $\vdash_{TT} pr(\ulcorner \phi \urcorner)$
  - Models proofs: P *iff* $\ulcorner P \urcorner$

- Correctness of encodings: faithfulness and adequacy

  Requires study of metatheory of metalogic: Are our encodings of FOL in $\lambda^{\rightarrow}$ more than just a syntactic trick?

First-Order Syntax with $\lambda^{\rightarrow}$

- Propositional logic

$$P ::= x \mid \neg P \mid P \wedge P \mid P \Rightarrow P \ldots$$

- Programming languages/algebraic specification

datatype *Prop* = *VarInject* of *Variable* | *not* of *Prop*
| *and* of *Prop**Prop | *imp* of *Prop**Prop

- $\lambda^{\rightarrow}$ approach
  - Type declarations for context $\mathcal{B} = \{o\}$
  - Signature types constants:

$$\Sigma = \{not : o \rightarrow o, and : o \rightarrow o \rightarrow o, imp : o \rightarrow o \rightarrow o\}$$

  - Context types propositional variables

First-Order Syntax (cont.)

- Example: $a : o \vdash imp(not\, a)a : o$

$$\dfrac{\dfrac{a : o \vdash imp : o \to o \to o \qquad \dfrac{a : o \vdash not : o \to o \quad a : o \vdash a : o}{a : o \vdash not\, a : o}}{a : o \vdash imp(not\, a) : o \to o} \qquad a : o \vdash a : o}{a : o \vdash imp(not\, a)a : o}$$

- Non example: $a : o \vdash not(imp\, a)a : o$

$$\dfrac{a : o \vdash not : o \to o \qquad \dfrac{a : o \vdash imp : o \to o \to o \quad a : o, \vdash a : o}{a : o \vdash imp\, a : o \to o}}{???}$$

No proof possible! (requires analysis of normal forms)

First-Order Syntax (cont.)

- Desire bijection $\ulcorner\cdot\urcorner : Prop \to o$

- Part 1: adequacy

$$p \in Prop \textit{ then } \Gamma \vdash \ulcorner p \urcorner : o$$
$$(\neg a) \Rightarrow b \in Prop \textit{ therefore } imp(not\, a)b : o$$

- Formalize mapping $\ulcorner\cdot\urcorner$

$$\begin{array}{lcl} \ulcorner x \urcorner & = & x \text{ for } x \text{ a variable} \\ \ulcorner \neg P \urcorner & = & not \ulcorner P \urcorner \\ \ulcorner P \wedge Q \urcorner & = & and \ulcorner P \urcorner \ulcorner Q \urcorner \end{array}$$

- Formal statement accounts for variables

  if $x \in FV(P) \Rightarrow x : o \in \Delta$ **and if** $P \in Prop$ **then** $\Delta \vdash \ulcorner P \urcorner : o$

- Proof of adequacy by induction on Prop

FOL/Syntactic Bijection (cont.)

- Part 2: faithfulness

$$\Delta \vdash t : o \textit{ then } \ulcorner t \urcorner^{-1} \in Prop$$

- Define $\ulcorner . \urcorner^{-1}$

$$\begin{array}{lcl} \ulcorner x \urcorner^{-1} & = & x \quad \textit{for } x \textit{ a variable} \\ \ulcorner not\, P \urcorner^{-1} & = & \neg \ulcorner P \urcorner^{-1} \\ \ulcorner and\, P\, Q \urcorner^{-1} & = & \ulcorner P \urcorner \wedge \ulcorner Q \urcorner \end{array}$$

- Trivially $\ulcorner \ulcorner p \urcorner \urcorner^{-1} = p$, but what about $\ulcorner \ulcorner t \urcorner^{-1} \urcorner = t$?

  $t = not\,((\lambda x^o.\, x)a)$, $t : o$, what is $\ulcorner t \urcorner^{-1}$?

Faithfulness (cont.)

- Problem: too many representatives in $\lambda^{\rightarrow}$, e.g. $\neg a$

$$\frac{a : o \vdash not : o \rightarrow o \quad a : o \vdash a : o}{a : o \vdash not\, a : o}\, app$$

$$\frac{a : o \vdash not : o \rightarrow o \qquad \dfrac{\dfrac{a : o, x : o \vdash x : o}{a : o \vdash \lambda x^{o}.\, x : o \rightarrow o}\, abs \qquad a : o \vdash a : o}{a : o \vdash (\lambda x^{o}.\, x)a : o}\, app}{a : o \vdash not\, ((\lambda x^{o}.\, x)a) : o}\, app$$

Faithfulness (cont.)

- If $t : o$, then $t =_{\beta\eta} t'$, for $t' : o$ a *canonical* ($\beta\eta$-long) normal form

$$
\begin{array}{lcl}
not\,((\lambda x.\,x)a) & =_{\beta\eta} & not\,a \\
not & =_{\beta\eta} & \lambda x.\,not\,x \\
imp\,(not\,((\lambda x.\,x)a)) & =_{\beta\eta} & \lambda x.\,imp\,(not\,a)\,x
\end{array}
$$

- **Theorem**: The encoding $\ulcorner\cdot\urcorner$ is a bijection between propositional formulae with free variables in Δ and canonical terms t', where $\Delta \vdash t' : o$

Faithfulness (cont.)

- **Proof**: Based on normalization

$$\dfrac{\dfrac{x : \sigma \vdash e : \tau}{\vdash \lambda x^{\sigma}.\, e \ : \sigma \to \tau}\, abs \qquad \vdash e' : \sigma}{\vdash (\lambda x^{\sigma}.\, e) e' \ : \tau}\, app$$

$$\Downarrow$$

$$\vdash e[x \leftarrow e'] : \tau$$

- **Corollary**: $t : o$ then $t =_{\beta\eta} t'$ and $\ulcorner t' \urcorner^{-1} \in Prop$ for some canonical t'

Problems with First-Order Syntax

- What about quantifiers ?

$$all : var \to o \to o \qquad \forall x.\, p \rightsquigarrow all\, x\, p$$

- First-order syntax requires explicit encoding of standard operations
  - binding: x bound in P in $\forall x.\, P \Leftrightarrow x$ bound in P in $all\, x\, P$
  - Substitution for bound variables:

$$\frac{\forall x.\, P_x}{P_t}\ \forall\text{-}E \qquad \frac{\dfrac{\forall x.\, x = x}{x = x[x \leftarrow 0]}\ \forall\text{-}E}{0 = 0}\ \text{Substitution}$$

  - Equivalence under bound variable renaming

$$(\forall x.\, P \Leftrightarrow \forall y.\, P[x \leftarrow y])$$

- Each requires explicit 'programming'

Higher-Order Abstract Syntax (HOAS)

- Example: first-order arithmetic (FOA)

$$
\begin{array}{lcl}
\textit{Terms } T & ::= & x \mid 0 \mid sT \mid T + T \mid T \times T \\
\textit{Formulae } F & ::= & T = T \mid \neg F \mid F \wedge F \mid \ldots \\
 & & \forall x.\, F \mid \exists x.\, F
\end{array}
$$

- Type declarations for context $\mathcal{B} = \{i, o\}$

- Signature $\Sigma = \Sigma_T \cup \Sigma_P \cup \Sigma_Q$:

$$
\begin{array}{lcl}
\Sigma_T & = & \{0 : i,\ s : i \to i,\ plus : i \to i \to i,\ times : i \to i \to i\} \\
\Sigma_P & = & \{eq : i \to i \to o,\ not : o \to o,\ and : o \to o \to o,\ \ldots\} \\
\Sigma_Q & = & \{all : (i \to o) \to o,\ exists : (i \to o) \to o\}
\end{array}
$$

HOAS (cont.)

- Faithfulness/adequacy: terms and formulae represented by (canonical) members of i and o

$$
\begin{array}{lcl}
0 + s0 & \Leftrightarrow & plus\,0\,(s0) \\
\forall x.\, x = x & \Leftrightarrow & all(\lambda x^{i}.\, eq\, x\, x) \\
\forall x.\, \exists y.\, \neg(x + x = y) & \Leftrightarrow & all(\lambda x^{i}.\, exists(\lambda y^{i}.\, not\,(eq\,(plus\, x\, x)\, y)))
\end{array}
$$

- Example derivation

$$
\dfrac{\vdash all : (i \to o) \to o \qquad \dfrac{\dfrac{\dfrac{x : i \vdash eq : i \to i \to o \quad x : i \vdash x : i}{x : i \vdash eq\, x : i \to o} \qquad x : i \vdash x : i}{x : i \vdash eq\, x\, x : o}}{\vdash \lambda x^{i}.\, eq\, x\, x : i \to o}}{\vdash all(\lambda x^{i}.\, eq\, x\, x) : o}
$$

HOAS — Why Higher Order Syntax?

- *Order*: For type τ written $\tau_1 \to \ldots \to \tau_n \to \tau_0$, right associated, $\tau_0 \in \mathcal{B}$:
  - $Ord(\tau) = 0$ if $\tau \in \mathcal{B}$
  - $Ord(\tau) = 1 + max(Ord(\tau_i))$,

- Term/propositional operators are first-order

$$and : o \to o \to o$$

- Variable binding operators are higher-order

$$all : (i \to o) \to o$$

- What is order of summation operator $sum : i \to i \to (i \to i) \to i$?

$$\sum_{x=0}^{n}(x+2) \quad \rightsquigarrow sum\, 0\, n\, (\lambda x^{i}.\, plus\, x\, (ss0))$$

HOAS — Why Abstract?

- Standard operations on syntax left implicit
  - binding: x bound in P in $\forall x.\, P \Leftrightarrow x$ bound in P in $all(\lambda x^i.\, P)$
  - Substitution for bound variables:

$$\frac{\forall x.\, P_x}{P_t}\,\forall\text{-}E \quad \Leftrightarrow \quad \frac{all(P)}{P(t)}\,\forall\text{-}E$$

$$\frac{\dfrac{\forall x.\, x = x}{x = x[x \leftarrow 0]}\,\forall\text{-}E}{0 = 0}\,\text{Substitution} \quad \Leftrightarrow \quad \frac{\dfrac{all(\lambda x^i.\, x = x)}{(\lambda x^i.\, x = x)0}\,\forall\text{-}E}{0 = 0}\,\beta\text{-reduction}$$

  - Equivalence under bound variable renaming

$$(\forall x.\, P \Leftrightarrow \forall y.\, P[x \leftarrow y]) \quad \Leftrightarrow \quad all(\lambda x^i.\, P) =_\alpha all(\lambda y^i.\, P[x \leftarrow y])$$

- $\lambda^{\rightarrow}$ implementation supports standard operations on syntax!

Summary of HOAS

Object Language	Meta Language
Syntactic Category Term, Prop	Type Declaration $\{i, o\} \in \mathcal{B}$
Variable x	Metalogic Variable x
Constructor $\wedge$	First-order Constant $and : o \rightarrow o \rightarrow o$
Binding Operator $\forall$	Second-order Constant $all : (i \rightarrow o) \rightarrow o$
Meaningful Expressions $a \wedge b \in Prop$	Members of Types $(and\, a\, b) : o$

Can $\lambda^{\rightarrow}$ adequately represent proofs?

- Typical rules for $Prop$ are:

$$\frac{A \wedge B}{A}\ \wedge\text{-}EL \qquad \frac{A \wedge B}{B}\ \wedge\text{-}ER \qquad \frac{A \quad B}{A \wedge B}\ \wedge\text{-}I$$

- Try ML-style typing with $pf \in \mathcal{B}$

$$\begin{array}{l} andel, ander : pf \rightarrow pf \\ andi : pf \rightarrow pf \rightarrow pf \end{array}$$

- Typing is too weak

$$andel(\ldots)(\ldots) : pf \ \textit{then} \ \ ander(\ldots)(\ldots) : pf$$

- Simple typing doesn't express dependencies

  Analogy to sorting: $\lambda x.x : A$ list $\rightarrow A$ list

Representing Proofs (cont.)

- Formulation with dependent types

$$pr : o \rightarrow Type \qquad pr(and\, a\, b) : Type$$

- Classify objects in levels: Term $\in$ Types $\in$ Kinds

$$pr \in o \rightarrow Type \in Kind$$

- Explicit quantification over types (new operator Π)

$$\Pi a^o b^o .\, pr(and\, a\, b) \rightarrow pr(a)$$

- Desired type theory corresponds to minimal logic over $\forall / \Rightarrow$ with ω-order quantification, known as the LF.

Further Reading

- Hindley and Seldin, Introduction to Combinators and λ-Calculus, Cambridge University Press, 1986.
- N.G. de Bruijn, "A Survey of the Project AUTOMATH", in Essays in Combinatory Logic, Lambda Calculus, and Formalism, Academic Press, 1980
- Harper, Honsell, and Plotkin, "A Framework for Defining Logics", JACM, January 1993.
- Avron, Honsell, Mason, Pollack, "Using Typed Lambda-Calculus to Implement Formal Systems on a Machine", JAR, 1992.

Metatheory II: Rules

David Basin

ETH Zurich
15.11.04

Overview — Proof Rules in the LF

- Last lecture: simple types classify syntax
- Dependent types classify rules: $pr : o \to Type$

$$\frac{A \wedge B}{A} \quad \rightsquigarrow \quad andel : \Pi x^o . \Pi y^o . pr(and\ x\ y) \to pr(x)$$

- Judgements as Types:

$$\begin{array}{c} \vdots\ P \\ \vdash \phi \end{array} \quad \rightsquigarrow \quad \ulcorner P \urcorner \in pr(\ulcorner \phi \urcorner)$$

  - Models syntax: $\phi \in Prop$ *iff* $\ulcorner \phi \urcorner \in o$
  - Models provability: $\vdash_L \phi$ *iff* $\vdash_{LF} pr(\ulcorner \phi \urcorner)$
  - Models proofs: P *iff* $\ulcorner P \urcorner$

Representing Proofs (cont.)

- Additional levels in the theory: Term $\in$ Type $\in$ Kind

$$pr \quad \in \quad o \rightarrow Type \quad \in \quad Kind$$

- Add dependent Π-types:

$$\lambda x^N.\, x + 1 : \Pi x^N.\, \{y : N | x < y\}$$

- Captures dependencies in proof rules

$andel : pr \rightarrow pr$ becomes $andel : \Pi a^o b^o.\, pr(and\, a\, b) \rightarrow pr(a)$

- Resulting type theory: minimal logic over $\forall/\Rightarrow$ with ω-order quantification

LF Presentation — Pseudo-Terms

- Let *Var* and *Const* be sets of variables and constants.

$$\{Type, Kind\} \in \mathit{Const}$$

- Pseudo-terms $\mathcal{T}$

$$\mathcal{T} ::= \mathit{Var} \mid \mathit{Const} \mid \mathcal{T}\mathcal{T} \mid \lambda x^{\mathcal{T}}.\mathcal{T} \mid \Pi x^{\mathcal{T}}.\mathcal{T}$$

  - In $\Pi x^{\tau_1}.\tau_2$, x is bound in τ_2
  - Write $\Pi x^{\tau_1}.\tau_2$ as $\tau_1 \rightarrow \tau_2$ when x not in τ_2

- Rules prescribe which pseudo-terms are sensible, e.g., in $\lambda^{\rightarrow}$
  - Some are terms: $\lambda x^N.x$
  - Some are types: $\Pi x^N.N \equiv N \rightarrow N$
  - Some are nonsense: $\Pi x^{\lambda y^N.y}.x$

LF — Signatures, $\vdash_{sig}$

- Let s range over sorts $\{Type, Kind\}$
- Define valid signature $\vdash_{sig}$, context $\vdash_{con}$, and type assignment $\vdash_{\Sigma}$
- A signature Σ is a sequence ($c \in$ *Const*)

$$\Sigma ::= \langle\,\rangle \mid \Sigma, c : A$$

- Valid signature Σ satisfies $\vdash_{sig}$ (all constants are well-typed or well-kinded)

$$\vdash_{sig} \langle\,\rangle \qquad \frac{\vdash_{sig} \Sigma \quad \vdash_{\Sigma} A : s \quad c \notin dom(\Sigma)}{\vdash_{sig} \Sigma, c : A}$$

- Example (type environment for constants):

$$\Sigma = \langle o : Type, not : o \rightarrow o, \ldots \rangle$$

LF Context, $\vdash_{con}$

- A context Γ is a sequence (for $x \in \mathit{Var}$)

$$\Gamma ::= \langle\,\rangle \;\mid\; \Gamma, x : A$$

- Γ valid w.r.t. Σ if it satisfies $\vdash_{con}$ (all variables are well-typed or well-kinded)

$$\vdash_{con} \langle\,\rangle \qquad \frac{\vdash_{con} \Gamma \quad \Gamma \vdash_{\Sigma} A : s \quad x \notin dom(\Gamma)}{\vdash_{con} \Gamma, x : A}$$

- Example ('symbol table for variables'):

$$\Gamma = \langle x : o, y : o, z : o \to o \rangle$$

LF Type Assignment Relation, $\vdash_\Sigma$

axiom $\Gamma \vdash_\Sigma \mathit{Type} : \mathit{Kind}$

assumption $$\frac{c : A \in \Sigma}{\Gamma \vdash_\Sigma c : A}$$

hypothesis $$\frac{x : A \in \Gamma}{\Gamma \vdash_\Sigma x : A}$$

formation $$\frac{\Gamma \vdash_\Sigma A : \mathit{Type} \quad \Gamma, x : A \vdash_\Sigma B : s}{\Gamma \vdash_\Sigma \Pi x^A . B : s}$$

abstraction $$\frac{\Gamma, x : A \vdash_\Sigma b : B \quad \Gamma \vdash_\Sigma \Pi x^A . B : s}{\Gamma \vdash_\Sigma \lambda x^A . b : \Pi x^A . B}$$

application $$\frac{\Gamma \vdash_\Sigma F : \Pi x^A . B \quad \Gamma \vdash_\Sigma a : A}{\Gamma \vdash_\Sigma F(a) : B[x \leftarrow a]}$$

Restriction of LF rules are $\lambda^{\rightarrow}$ rules

- Restrict typing assertion $e : B$

  where e is $\lambda^{\rightarrow}$ term and B is simple-type.

- LF rules formalize well-formed signature, context, and type.

$\tau ::= T \mid \tau \rightarrow \tau$ (Type Grammar)

$$\Gamma, x : \tau, \Delta \vdash x : \tau \quad \text{hyp}$$

$$\frac{\Gamma \vdash e : \sigma \rightarrow \tau \quad \Gamma \vdash e' : \sigma}{\Gamma \vdash ee' : \tau} \text{ app} \qquad \frac{\Gamma, x : \sigma \vdash e : \tau}{\Gamma \vdash \lambda x^{\sigma}.\, e \; : \sigma \rightarrow \tau} \text{ abs}$$

LF and $\lambda^{\rightarrow}$ — Example

- Signature for logic over arrow (base type explicit):

$$\Sigma = \langle o : Type, not : o \rightarrow o, imp : o \rightarrow o \rightarrow o \rangle \,,$$

- Signature well-formed

$$\vdash_{\langle\,\rangle} Type : Kind$$

and

$$\frac{\dfrac{o : Type \in \langle o : Type \rangle}{\vdash_{\langle o:Type \rangle} o : Type} \qquad \dfrac{o : Type \in \langle o : Type \rangle}{x : o \vdash_{\langle o:Type \rangle} o : Type}}{\vdash_{\langle o:Type \rangle} o \rightarrow o : Type}$$

similar for type of imp

(Full) LF Type Theory

- Additional rules add power, e.g., formation (here $s = Kind$)

$$\frac{\Gamma \vdash_\Sigma A : Type \quad \Gamma, x : A \vdash_\Sigma B : Kind}{\Gamma \vdash_\Sigma \Pi x^A . B : Kind}$$

$$\begin{aligned} \Sigma_{prop} \quad = \quad & \langle o : Type, pr : o \to Type, not : o \to o, \\ & imp : o \to o \to o, and : o \to o \to o \rangle \end{aligned}$$

- Type-valued functions in context are now well-typed

$$\frac{o : Type \vdash_\Sigma o : Type \quad o : Type, x : o \vdash_\Sigma Type : Kind}{o : Type \vdash_\Sigma o \to Type : Kind}$$

- Moreover, dependent types not possible in $\lambda^{\to}$

Example of Dependent Types — Rule Formalization

- Consider $andel : \Pi x^o.\, \Pi y^o.\, pr(and\ x\ y) \to pr(x)$

- Part I

$$\dfrac{x : o, y : o \vdash_\Sigma pr : o \to Type \qquad \dfrac{\dfrac{x : o, y : o \vdash_\Sigma and : o \to o \to o \quad x : o, y : o \vdash_\Sigma x : o}{x : o, y : o \vdash_\Sigma and\, x : o \to o}\,app \qquad x : o, y : o \vdash_\Sigma y : o}{x : o, y : o \vdash_\Sigma and\, x\, y : o}\,app}{x : o, y : o \vdash_\Sigma pr(and\, x\, y) : Type}\,app$$

- Part II

$$\dfrac{\vdash_\Sigma o : Type \qquad \dfrac{x : o \vdash_\Sigma o : Type \qquad \dfrac{\text{(Part I)} \qquad \dfrac{x : o, y : o, z : pr(x) \vdash_\Sigma pr : o \to Type \quad x : o, y : o, z : pr(x) \vdash_\Sigma x : o}{x : o, y : o, z : pr(x) \vdash_\Sigma pr(x) : Type}\,app}{x : o, y : o \vdash_\Sigma pr(and\, x\, y) \to pr(x) : Type}\,form}{x : o \vdash_\Sigma \Pi y^o.\, pr(and\, x\, y) \to pr(x) : Type}\,form}{\vdash_\Sigma \Pi x^o.\, \Pi y^o.\, pr(and\, x\, y) \to pr(x) : Type}\,form$$

Metatheoretic Properties of LF

- $\forall/\Rightarrow$ fragment of intuitionistic ω-order logic

$$\Pi x^o.\, \Pi y^o.\, pr(and\ x\ y) \rightarrow pr(x)$$
$$\Downarrow$$
$$\forall x^o.\, \forall y^o.\, pr(and\ x\ y) \Rightarrow pr(x)$$

- Unique normal forms and subject reduction

  Faithfulness/adequacy by analysis of normal forms

- $\Gamma \vdash_\Sigma t : \tau$ is decidable.
  - Well-formedness is decidable: $\tau : Type$ or $\tau : Kind$
  - Proof checking is decidable: $t : \tau$

Idea of LF Encodings

Rules organized around basic assertions (judgements)

- $pr(p)$: p is provable
- $valid(p)$: p is true in all models
- $true(p)$: p is true in some model
- $acc(w_1, w_2)$: '$world$' w_1 accesses w_2
- Multiple judgements, e.g. type theory and modal logics

+ higher-order judgements: (J_i judgement, C syntactic category)

- Schematic: $\bigwedge_{x \in C} J(x)$
- Hypothetical: $J_1 \vdash_\Sigma J_2$ expresses consequence and allows hybrid notions of consequence

$$valid(p) \vdash_\Sigma true(p)$$

Idea of LF Encodings: *Judgements as Types*

- Basic judgements associated with type families,
  e.g. $pr : o \rightarrow Type$

$$\overline{p \text{ provable}} \rightsquigarrow pr(p)$$

- Higher-order forms available

$$\overline{\bigwedge_{x \in C} J(x)} \rightsquigarrow \Pi x^{\overline{C}}.\, \overline{J(x)}$$

$$\overline{J_1 \vdash J_2} \rightsquigarrow \overline{J_1} \rightarrow \overline{J_2}$$

- Truth = inhabitation

$t : pr(p)$ *iff* t represents proof that p is provable

Example: Propositional Logic

- Formalize language ....

$$P ::= x \mid P \wedge P \mid P \Rightarrow P \ldots$$

- ... and proof rules

$$\frac{A \wedge B}{A}\ \wedge\text{-}EL \qquad \frac{A \wedge B}{B}\ \wedge\text{-}ER \qquad \frac{A \quad B}{A \wedge B}\ \wedge\text{-}I$$

$$\frac{A \Rightarrow B \quad A}{B}\ \Rightarrow\text{-}E \qquad \frac{\begin{array}{c}[A]\\ \vdots \\ B\end{array}}{A \Rightarrow B}\ \Rightarrow\text{-}I$$

Propositional Logic in LF

- Σ contains syntactic categories

$$o : Type$$

- and judgements

$$pr : o \rightarrow Type$$

- and operators (declaring syntax)

$$imp : o \rightarrow o \rightarrow o, and : o \rightarrow o \rightarrow o, \ldots$$

- and rules (next page)

Propositional Logic in LF

- and-rules (and impe) are 1st-order judgements

$$\frac{A \wedge B}{A} \quad \rightsquigarrow \quad \Pi a^o b^o.\, pr(and\ a\ b) \rightarrow pr(a)$$

- $\Rightarrow$-I is 2nd-order judgement

$$\frac{\begin{array}{c}[A] \\ \vdots \\ B\end{array}}{A \Rightarrow B} \Rightarrow\text{-}I \quad \rightsquigarrow \quad \begin{array}{l}\Pi a^o b^o.\, (pr(a) \rightarrow pr(b)) \\ \quad \rightarrow pr(imp\ a\ b)\end{array}$$

  - Consequence over *hypothetical proofs*: $(A \vdash B) \vdash A \Rightarrow B$
  - Shifts assumption discharging to LF

$$\frac{\dfrac{x : pr(a) \vdash_\Sigma \ldots : pr(b)}{\vdash_\Sigma \ldots : pr(a) \rightarrow pr(b)}\, abs_{T,T}}{\vdash_\Sigma \ldots : pr(imp\ a\ b)}\, impi^*$$

Propositional Logic — More Rules

$$\frac{A}{A \vee B}\ \vee\text{-}IL \qquad \frac{B}{A \vee B}\ \vee\text{-}IR \qquad \frac{A \vee B \quad \begin{array}{c}[A]\\ \vdots \\ C\end{array} \quad \begin{array}{c}[B]\\ \vdots \\ C\end{array}}{C}\ \vee\text{-}E$$

- Augment operators: $or : o \to o \to o$

- Augment rules:

$$\begin{aligned}
&oril : \Pi a^o b^o.\, pr(a) \to pr(or\, a\, b)\\
&orir : \Pi a^o b^o.\, pr(b) \to pr(or\, a\, b)\\
&ore : \Pi a^o b^o c^o.\, pr(or\, a\, b) \to (pr(a) \to pr(c))\\
&\qquad \to (pr(b) \to pr(c)) \to pr(c)
\end{aligned}$$

Propositional Proofs: $a \Rightarrow a$

Part I:

$$\dfrac{\dfrac{a : o \vdash_\Sigma impi : \Pi a^o b^o.\, (pr(a) \to pr(b)) \to pr(imp\, a\, b) \quad a : o \vdash_\Sigma a : o}{a : o \vdash_\Sigma impi\, a : \Pi b^o.\, (pr(a) \to pr(b)) \to pr(imp\, a\, b)} \quad a : o \vdash_\Sigma a : o}{a : o \vdash_\Sigma impi\, a\, a : (pr(a) \to pr(a)) \to pr(imp\, a\, a)}$$

Part II:

$$\dfrac{a : o, y : pr(a) \vdash_\Sigma y : pr(a)}{a : o \vdash_\Sigma \lambda y^{pr(a)}.\, y : pr(a) \to pr(a)}$$

Part III:

$$\dfrac{\dfrac{\text{(Part I)} \qquad \text{(Part II)}}{a : o \vdash_\Sigma impi\, a\, a\, (\lambda y^{pr(a)}.\, y) : pr(imp\, a\, a)}}{\vdash_\Sigma \lambda a^o.\, impi\, a\, a\, (\lambda y^{pr(a)}.\, y) : \Pi a^o.\, pr(imp\, a\, a)}$$

LF — Aside on System Support

Proof detail impractical. Implementations support:

- Implicit quantification/type reconstruction

$$\lambda a^o.\, impi\, a\, a\, (\lambda y^{pr(a)}.\, y) : \Pi a^o.\, pr(imp\, a\, a)$$

- Automatic type checking
- "Resolution" and refinement style proof.

$$\frac{\Gamma \vdash_\Sigma \ldots : pr(imp\, a\, a)}{\Gamma, x : pr(a) \vdash_\Sigma \ldots : pr(a)}\, impi^*$$

- Term synthesis via unification

$$\Gamma \vdash_\Sigma ?t : pr(imp\, a\, a)$$

- Pretty printing, implicit contexts, and implicit judgement coercions

$$\frac{\vdash a \Rightarrow a}{a \vdash a}$$

- Tactics

LF — Terms Encode Proofs

- $a \Rightarrow a \quad \rightsquigarrow \quad \Pi a^o.\, pr(imp\, a\, a)$

$$\lambda a^o.\, impi\, a\, a\, (\lambda y^{pr(a)}.\, y)$$

$$\frac{[A]^y}{A \Rightarrow A}\ \Rightarrow\text{-}I^y$$

- $a \Rightarrow b \Rightarrow a \quad \rightsquigarrow \quad \Pi a^o b^o.\, pr(imp\, a\, (imp\, b\, a))$

$$\lambda a^o.\, \lambda b^o.\, impi\, a\, (imp\, b\, a)\, (\lambda x^{pr(a)}.\, impi\, b\, a\, (\lambda y^{pr(b)}.\, x))$$

$$\dfrac{\dfrac{[A]^x}{B \Rightarrow A}\ \Rightarrow\text{-}I^y}{A \Rightarrow (B \Rightarrow A)}\ \Rightarrow\text{-}I^x$$

- Adequacy & faithfulness: There exists a bijection between proofs of $\phi \in Prop$ and canonical members of $pr(\phi)$

Schematic Judgements — F.O. Arithmetic Example

- Review: syntax

$$\begin{array}{lcl} \textit{Terms}\ \ T & ::= & x \mid 0 \mid sT \mid T+T \mid T \times T \\ \textit{Formulae}\ \ F & ::= & t = t \mid \neg F \mid F \wedge F \mid \ldots \\ & & \forall x.\, A \mid \exists x.\, A \end{array}$$

- Type declarations for context

$$B = i : \textit{Type},\ o : \textit{Type}$$

- Context $\Gamma = \langle B, \Gamma_T, \Gamma_P, \Gamma_Q \rangle$:

$$\begin{array}{lcl} \Gamma_T & = & 0 : i,\ s : i \to i,\ plus : i \to i \to i,\ times : i \to i \to i \\ \Gamma_P & = & eq : i \to i \to o,\ not : o \to o,\ and : o \to o \to o,\ \ldots \\ \Gamma_Q & = & all : (i \to o) \to o,\ exists : (i \to o) \to o \end{array}$$

FO Arithmetic — ∀-Quantifier Rules

$$\frac{A}{\forall x.\, A}\ \forall\text{-}I^{*} \qquad \frac{\forall x.\, A}{A[x \leftarrow t]}\ \forall\text{-}E$$

$*$: x not free in any undischarged assumptions on which A depends

$$alli : \Pi A^{i \to o}.\, (\Pi x^{i}.\, pr(A(x))) \to pr(all(A))$$
$$alle : \Pi A^{i \to o} x^{i}.\, pr(all(A)) \to pr(A(x))$$

Side Conditions Captured via Π-binding

Prove $(\forall x.\, A \Rightarrow B(x)) \Rightarrow (A \Rightarrow \forall x.\, B(x))$ in context $\Gamma, A : o, B : i \to o, x : i$

Part I:

$$\frac{pr(all(\lambda x^i.\, A \Rightarrow B(x))), pr(A) \vdash_\Sigma pr(all(\lambda x^i.\, A \Rightarrow B(x)))}{pr(all(\lambda x^i.\, A \Rightarrow B(x))), pr(A) \vdash_\Sigma pr(A \Rightarrow B(x))}\ alle$$

Part II:

$$\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{\text{(Part I)}\quad pr(all(\lambda x^i.\, A \Rightarrow B(x))), pr(A) \vdash_\Sigma pr(A)}{pr(all(\lambda x^i.\, A \Rightarrow B(x))), pr(A) \vdash_\Sigma pr(B(x))}\ impe}{pr(all(\lambda x^i.\, (A \Rightarrow B(x)))), pr(A) \vdash_\Sigma \Pi x^i.\, pr(B(x))}\ abs\ \text{(discharge } x : i\text{)}}{pr(all(\lambda x^i.\, (A \Rightarrow B(x)))), pr(A) \vdash_\Sigma pr(all(\lambda x^i.\, B(x)))}\ alli}{pr(all(\lambda x^i.\, (A \Rightarrow B(x)))) \vdash_\Sigma pr(A \Rightarrow all(\lambda x^i.\, B(x)))}\ impi}{pr(all(\lambda x^i.\, (A \Rightarrow B(x)))) \Rightarrow (A \Rightarrow all(\lambda x^i.\, B(x))))}\ impi$$

FO Arithmetic — Substitution

$$\frac{t = u \quad A[t]}{A[u]}\, sub \qquad \frac{a = b \quad P(a, a, a)}{P(a, b, a)}$$

$A[t]$ means some distinguished (not necessarily all) occurrences of t in A. In above, $A = \lambda x.\, P(a, x, a)$

$$sub : \Pi t^i u^i A^{i \to o}.\, pr(eq\, t\, u) \to pr(A(t)) \to pr(A(u))$$

Aside on Derived Rules

- Encoding of FOA adequate and faithful

  But more then FOA sentences are expressible/derivable!

- Rules are derivable

$$\frac{\forall x.\, A \qquad \begin{array}{c}[A[x \leftarrow t]] \\ \vdots \\ B\end{array}}{B}\, alle_{SH}$$

$$alle_{SH} : \Pi A^{i \to o} B^{o}.\, pr(all(A)) \to ((\Pi x^{i}.\, pr(A(x))) \to pr(B)) \to pr(B)$$

- Inhabitants are proof-construction functions. $alle_{SH} =$

$$\lambda A^{i \to o} B^{o} p^{pr(all(A))} q^{\Pi x^{i}.\, pr(A(x)) \to pr(B)}.\, q(\lambda x^{i}.\, alle(\lambda x^{i}.\, B(x)(x)(p)))$$

- Can verify all *derivable* rules.

  Judgements are open: no induction over encoded theory

Summary of Proof/Rule Encoding

Object Language	Meta Language
Judgement $\vdash_\Sigma p$	Type Family $pr : o \rightarrow Type$
Inference Rule $\dfrac{A \wedge B}{A} \wedge\text{-}EL$	Constant Declaration $andel : \Pi a^o b^o . pr(and\, a\, b) \rightarrow pr(a)$
Deduction	Typing Proof
Deductive System	Context Declaration

Advantages/Disadvantages of LF

+ High-level representation: HOAS and assumption management for ND

+ Judgements as Types: Proof construction and adequacy arguments

+ Good metatheoretic properties

  + Decidable type checking $\Gamma \vdash_{\Sigma} t : \tau$

+/− Supports some metatheoretic extensibility

  − No rules by induction over theory
  + Derived rules hold in theory extensions

+/− Biased towards certain logics: $\vdash_L$ shares properties with $\vdash_{LF}$

Relation to Type Theories based on HOL (Isabelle)

- $\lambda^{\rightarrow}$ used to declare syntax

$$\begin{aligned}\Gamma_{prop} &= \langle o : Type,\, pr : o \rightarrow Prop,\, not : o \rightarrow o, \\ &\quad imp : o \rightarrow o \rightarrow o, and : o \rightarrow o \rightarrow o\rangle\end{aligned}$$

- $\forall/\Rightarrow$ fragment of HOL used to declare axioms

$$andel : \Pi x^o.\, \Pi y^o.\, pr(and\ x\ y) \rightarrow pr(x)$$
$$\downarrow$$
$$andel : \forall x^o.\, \forall y^o.\, pr(and\ x\ y) \Rightarrow pr(x)$$

- Formalizations similar, but no Judgements as Types

- Adequacy/faithfulness by proof (not term) normalization

Further Reading

- Barendregt, "Introduction to Generalized Type Systems", Journal of Functional Programming, April 1991.
- Harper, Honsell, and Plotkin, "A Framework for Defining Logics", JACM, January 1993.
- Paulson, "Isabelle: A Generic Theorem Prover", Springer LNCS 828.

Isabelle: Resolution

Burkhart Wolff

Higher-Order Unification

The typed λ-calculus is the syntactic device of Isabelle's built-in metalogic.

Higher-Order Unification

The typed λ-calculus is the syntactic device of Isabelle's built-in metalogic.The used version of the λ-calculus contains also "metavariables".

Higher-Order Unification

The typed λ-calculus is the syntactic device of Isabelle's built-in metalogic.The used version of the λ-calculus contains also "metavariables". Logically equivalent to free variables, they can be arbitrarily instantiated by terms or formulas.

Higher-Order Unification

The typed λ-calculus is the syntactic device of Isabelle's built-in metalogic.The used version of the λ-calculus contains also "metavariables". Logically equivalent to free variables, they can be arbitrarily instantiated by terms or formulas. They can be viewed as "delayed substitutions", that were provided during a proof at need.

Higher-Order Unification

The typed λ-calculus is the syntactic device of Isabelle's built-in metalogic.The used version of the λ-calculus contains also "metavariables". Logically equivalent to free variables, they can be arbitrarily instantiated by terms or formulas. They can be viewed as "delayed substitutions", that were provided during a proof at need.

This means: Isabelle´s proof engine will

Higher-Order Unification

The typed λ-calculus is the syntactic device of Isabelle's built-in metalogic.The used version of the λ-calculus contains also "metavariables". Logically equivalent to free variables, they can be arbitrarily instantiated by terms or formulas. They can be viewed as "delayed substitutions", that were provided during a proof at need.

This means: Isabelle´s proof engine will

- rename metavariables

Higher-Order Unification

The typed λ-calculus is the syntactic device of Isabelle's built-in metalogic.The used version of the λ-calculus contains also "metavariables". Logically equivalent to free variables, they can be arbitrarily instantiated by terms or formulas. They can be viewed as "delayed substitutions", that were provided during a proof at need.

This means: Isabelle´s proof engine will

- rename metavariables
- unify metavariables

during rule application.

What is Higher-Order Unification?

Unification of terms e, e':

find substitution θ for metavariables such that

$\theta(e) =_{\alpha\beta\eta} \theta(e')$.

Examples:

$$\begin{aligned} ?X + ?Y &=_{\alpha\beta\eta} x + x \\ ?P(x) &=_{\alpha\beta\eta} x + x \\ f(?X\,x) &=_{\alpha\beta\eta} ?Y\,x \end{aligned}$$

What is Higher-Order Unification?

Unification of terms e, e':

find substitution θ for metavariables such that

$\theta(e) =_{\alpha\beta\eta} \theta(e')$.

Examples:

$$\begin{aligned} ?X + ?Y &=_{\alpha\beta\eta} x + x \\ ?P(x) &=_{\alpha\beta\eta} x + x \\ f(?X\,x) &=_{\alpha\beta\eta} ?Y\,x \end{aligned}$$

Why higher-order? Metavariables may be instantiated to functions, e.g. $[?P \leftarrow \lambda y.y + y]$.

Higher-Order Unification: Some Facts

- Unification modulo $\alpha\beta\eta$ (HO-unification) is semi-decidable for the (monomorphic) typed λ-calculus.

Higher-Order Unification: Some Facts

- Unification modulo $\alpha\beta\eta$ (HO-unification) is semi-decidable for the (monomorphic) typed λ-calculus.
- Isabelle´s Unification modulo $\alpha\beta\eta$ is incomplete.

Higher-Order Unification: Some Facts

- Unification modulo $\alpha\beta\eta$ (HO-unification) is semi-decidable for the (monomorphic) typed λ-calculus.
- Isabelle´s Unification modulo $\alpha\beta\eta$ is incomplete.
- HO-unification is well-behaved for most practical cases.

Higher-Order Unification: Some Facts

- Unification modulo $\alpha\beta\eta$ (HO-unification) is semi-decidable for the (monomorphic) typed λ-calculus.
- Isabelle´s Unification modulo $\alpha\beta\eta$ is incomplete.
- HO-unification is well-behaved for most practical cases.
- Important fragments (like HO-patterns) are decidable.

Higher-Order Unification: Some Facts

- Unification modulo $\alpha\beta\eta$ (HO-unification) is semi-decidable for the (monomorphic) typed λ-calculus.
- Isabelle´s Unification modulo $\alpha\beta\eta$ is incomplete.
- HO-unification is well-behaved for most practical cases.
- Important fragments (like HO-patterns) are decidable.
- HO-unification has possibly infinitely many solutions.

Three Sections on Deduction Techniques: Outline

- Resolution

Three Sections on Deduction Techniques: Outline

- Resolution
- Proof search

Three Sections on Deduction Techniques: Outline

- Resolution
- Proof search
- Term rewriting

Three Sections on Deduction Techniques: Outline

- Resolution
- Proof search
- Term rewriting

Resolution

Resolution is the basic mechanism for transforming proof states in Isabelle in order to construct a proof.

It involves unifying a certain part of the current goal (state) with a certain part of a rule, and replacing that part of the current goal.

We now look at several variants of resolution.

Resolution (`rule`, as in Prolog)

$$\frac{\phi_1 \quad \cdots \quad \phi_i \quad \cdots \quad \phi_n}{\psi}$$

$\phi_1, \ldots, \phi_n$ are current subgoals and ψ is original goal. Isabelle displays

`Level ...` (n `subgoals`)

ψ

1. ϕ_1

⋮

n. ϕ_n

Resolution (`rule`, as in Prolog)

$$\frac{\alpha_1 \cdots \alpha_m}{\beta}$$

$$\frac{\phi_1 \quad \cdots \quad \phi_i \quad \cdots \quad \phi_n}{\psi}$$

$\phi_1, \ldots, \phi_n$ are current subgoals and ψ is original goal. Isabelle displays

`Level ...` (n `subgoals`)

ψ

1. ϕ_1

   $\vdots$

n. ϕ_n

$[\![\alpha_1; \ldots; \alpha_m]\!] \Longrightarrow \beta$ is rule.

Resolution (`rule`, as in Prolog)

$$\frac{\alpha_1 \cdots \alpha_m}{\beta}$$

$$\frac{\phi_1 \quad \cdots \quad \phi_i \quad \cdots \quad \phi_n}{\psi}$$

Simple scenario where ϕ_i has no premises. Now β must be unifiable with selected subgoal ϕ_i.

Resolution (`rule`, as in Prolog)

$$\frac{\alpha'_1 \cdots \alpha'_m}{\beta'}$$

$$\frac{\phi'_1 \quad \cdots \quad \phi'_i \quad \cdots \quad \phi'_n}{\psi'}$$

Simple scenario where ϕ_i has no premises. Now β must be unifiable with selected subgoal ϕ_i.

We apply the unifier ($'$)

Resolution (`rule`, as in Prolog)

$$\frac{\phi'_1 \cdots \alpha'_1 \cdots \alpha'_m \cdots \phi'_n}{\psi'}$$

Simple scenario where ϕ_i has no premises. Now β must be unifiable with selected subgoal ϕ_i.

We apply the unifier ($'$)

We replace ϕ'_i by the premises of the rule.

Resolution (with Lifting over Parameters)

$$\frac{\phi_1 \quad \cdots \quad \bigwedge x.\phi_i \quad \cdots \quad \phi_n}{\psi}$$

Now suppose the i'th (selected) subgoal is preceded by $\bigwedge$ (metalevel universal quantifier).

Resolution (with Lifting over Parameters)

$$\frac{\alpha_1 \quad \cdots \alpha_m}{\beta}$$

$$\frac{\phi_1 \quad \cdots \quad \bigwedge x.\phi_i \quad \cdots \quad \phi_n}{\psi}$$

Rule

Resolution (with Lifting over Parameters)

$$\bigwedge x. \frac{\alpha_1(x) \cdots \alpha_m(x)}{\beta(x)}$$

$$\frac{\phi_1 \quad \cdots \quad \bigwedge x.\phi_i \quad \cdots \quad \phi_n}{\psi}$$

Rule is lifted over x: Apply $[?X \leftarrow ?X(x)]$.

Resolution (with Lifting over Parameters)

$$\frac{\phi_1 \quad \cdots \quad \bigwedge x.\phi_i \quad \cdots \quad \phi_n}{\psi} \qquad \bigwedge x.\frac{\alpha_1(x)\cdots\alpha_m(x)}{\beta(x)}$$

Rule is lifted over x: Apply $[?X \leftarrow ?X(x)]$.

As before, β must be unifiable with ϕ_i;

Resolution (with Lifting over Parameters)

$$\frac{\phi'_1 \quad \cdots \quad \bigwedge x.\phi'_i \quad \cdots \quad \phi'_n}{\psi'} \qquad \bigwedge x.\frac{\alpha'_1(x)\cdots\alpha'_m(x)}{\beta'(x)}$$

Rule is lifted over x: Apply $[?X \leftarrow ?X(x)]$.

As before, β must be unifiable with ϕ_i; apply the unifier.

Resolution (with Lifting over Parameters)

$$\frac{\phi'_1 \cdots \bigwedge x.\alpha'_1[x] \cdots \bigwedge x.\alpha'_m[x] \cdots \phi'_n}{\psi'}$$

Rule is lifted over x: Apply $[?X \leftarrow ?X(x)]$.

As before, β must be unifiable with ϕ_i; apply the unifier.

We replace ϕ'_i by the premises of the rule. $\alpha'_1, \ldots, \alpha'_m$ are preceded by $\bigwedge x$.

Resolution (with Lifting over Assumptions)

$$\frac{\phi_1 \quad \cdots \quad \begin{array}{c}[\phi_{i1} \cdots \phi_{ik_i}] \\ \vdots \\ \phi_i\end{array} \quad \cdots \quad \phi_n}{\psi}$$

Now, suppose the i'th (selected) subgoal has assumptions $\phi_{i1}, \ldots, \phi_{ik_i}$.

Resolution (with Lifting over Assumptions)

$$\frac{\phi_1 \quad \cdots \quad \begin{array}{c}[\phi_{i1} \cdots \phi_{ik_i}] \\ \vdots \\ \phi_i\end{array} \quad \cdots \quad \phi_n}{\psi} \qquad \frac{\alpha_1 \quad \cdots \quad \alpha_m}{\beta}$$

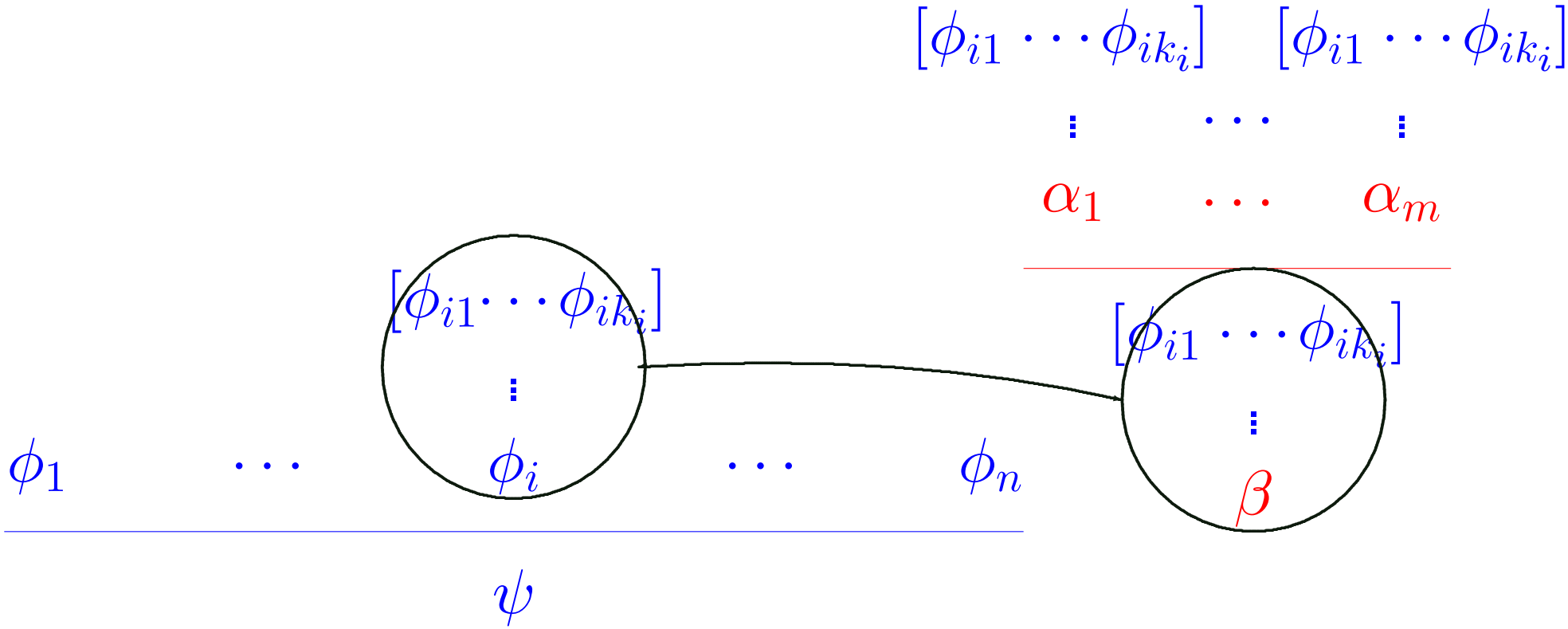
As before, we have a rule. In general β is not unifiable with the i'th subgoal, even assuming that β is unifiable with ϕ_i.

Resolution (with Lifting over Assumptions)

$$\frac{\phi_1 \quad \cdots \quad \begin{array}{c}[\phi_{i1} \cdots \phi_{ik_i}] \\ \vdots \\ \phi_i\end{array} \quad \cdots \quad \phi_n}{\psi} \qquad \frac{\begin{array}{c}[\phi_{i1} \cdots \phi_{ik_i}] \\ \vdots \\ \alpha_1\end{array} \quad \cdots \quad \begin{array}{c}[\phi_{i1} \cdots \phi_{ik_i}] \\ \vdots \\ \alpha_m\end{array}}{\begin{array}{c}[\phi_{i1} \cdots \phi_{ik_i}] \\ \vdots \\ \beta\end{array}}$$

Rule must be lifted over assumptions. No unification so far!

Resolution (with Lifting over Assumptions)



Now, subgoal and rule conclusion (below the bar) are unifiable.

Resolution (with Lifting over Assumptions)

$$\dfrac{\begin{array}{ccc} [\phi_{i1} \cdots \phi_{ik_i}] & \cdots & [\phi_{i1} \cdots \phi_{ik_i}] \\ \vdots & & \vdots \\ \alpha_1 & \cdots & \alpha_m \end{array}}{[\phi_{i1} \cdots \phi_{ik_i}]}$$

$$\dfrac{\phi_1 \quad \cdots \quad \begin{array}{c} [\phi_{i1} \cdots \phi_{ik_i}] \\ \vdots \\ \phi_i \end{array} \quad \cdots \quad \phi_n}{\psi} \qquad \begin{array}{c} [\phi_{i1} \cdots \phi_{ik_i}] \\ \vdots \\ \beta \end{array}$$

Now, subgoal and rule conclusion (below the bar) are unifiable.
Non-trivially, β must be unifiable with ϕ_i.

Resolution (with Lifting over Assumptions)

$$\frac{\phi'_1 \quad \cdots \quad \begin{array}{c}[\phi'_{i1} \cdots \phi'_{ik_i}] \\ \vdots \\ \phi'_i\end{array} \quad \cdots \quad \phi'_n}{\psi'} \qquad \frac{\begin{array}{c}[\phi'_{i1} \cdots \phi'_{ik_i}] \\ \vdots \\ \alpha'_1\end{array} \quad \cdots \quad \begin{array}{c}[\phi'_{i1} \cdots \phi'_{ik_i}] \\ \vdots \\ \alpha'_m\end{array}}{\begin{array}{c}[\phi'_{i1} \cdots \phi'_{ik_i}] \\ \vdots \\ \beta'\end{array}}$$

We apply the unifier.

Resolution (with Lifting over Assumptions)

$$\frac{\begin{array}{ccccc} & [\phi'_{i1} \cdots \phi'_{ik_i}] & & [\phi'_{i1} \cdots \phi'_{ik_i}] & \\ & \vdots & \cdots & \vdots & \\ \phi'_1 \cdots \phi'_{i-1} & \alpha'_1 & \cdots & \alpha'_m & \phi'_{i+1} \cdots \phi'_n \end{array}}{\psi'}$$

We replace the subgoal.

Rule Premises Containing $\Longrightarrow$

$$\frac{\phi'_1 \cdots \quad \begin{array}{c}[\phi'_{i1} \cdots \phi'_{ik_i}] \\ \vdots \\ \alpha'_j\end{array} \quad \cdots \phi'_n}{\psi'}$$

What if some α_j has the form $[\![\gamma_1; \ldots; \gamma_l]\!] \Longrightarrow \delta$?

Rule Premises Containing $\Longrightarrow$

$$\frac{\begin{array}{c}[\phi'_{i1} \ \cdots \ \phi'_{ik_i}] \\ \vdots \\ \phi'_1 \ \cdots \ [\![\gamma'_1; \ldots ; \gamma'_l]\!] \Longrightarrow \delta' \ \cdots \ \phi'_n\end{array}}{\psi'}$$

Is this what we get?

Rule Premises Containing $\Longrightarrow$

$$\frac{\phi'_1 \cdots \quad \begin{array}{c}[\phi'_{i1} \cdots \phi'_{ik_i}\ \gamma'_1 \cdots \gamma'_l] \\ \vdots \\ \delta'\end{array} \quad \cdots \phi'_n}{\psi'}$$

Is this what we get?

Well, we write $\vdots$ for $\Longrightarrow$, and use

$A \Longrightarrow B \Longrightarrow C \equiv [\![A; B]\!] \Longrightarrow C$.

Elimination-Resolution

$$\frac{\alpha_1 \cdots \alpha_m}{\beta}$$

$$\frac{\begin{array}{c}[\phi_{i1} \cdots \quad \phi_{il} \quad \cdots \phi_{ik_i}] \\ \vdots \\ \phi_1 \qquad \cdots \qquad \phi_i \qquad \cdots \qquad \phi_n\end{array}}{\psi}$$

Same scenario as before

Elimination-Resolution

$$\frac{\phi_1 \quad \cdots \quad \begin{matrix}[\phi_{i1} \cdots \phi_{il} \cdots \phi_{ik_i}] \\ \vdots \\ \phi_i\end{matrix} \quad \cdots \quad \phi_n}{\psi} \qquad \frac{\alpha_1 \cdots \alpha_m}{\beta}$$

Same scenario as before, but now β must be unifiable with ϕ_i, and α_1 must be unifiable with ϕ_{il}, for some l.

Elimination-Resolution

$$\frac{\phi'_1 \quad \cdots \quad \begin{array}{c}[\phi'_{i1} \cdots \phi'_{il} \cdots \phi'_{ik_i}] \\ \vdots \\ \phi'_i\end{array} \quad \cdots \quad \phi'_n \qquad \frac{\alpha'_1 \cdots \alpha'_m}{\beta'}}{\psi'}$$

Same scenario as before, but now β must be unifiable with ϕ_i, and α_1 must be unifiable with ϕ_{il}, for some l.

Apply the unifier.

Elimination-Resolution

$$\frac{\begin{array}{c}[\phi'_{i1} \cdots \phi'_{i,l-1}, \phi'_{i,l+1} \cdots \phi'_{ik_i}] \\ \vdots \\ \phi'_1 \cdots \phi'_{i-1} \ \alpha'_2\end{array} \quad \cdots \quad \begin{array}{c}[\phi'_{i1} \cdots \phi'_{i,l-1}, \phi'_{i,l+1} \cdots \phi'_{ik_i}] \\ \vdots \\ \alpha'_m \ \phi'_{i+1} \cdots \phi'_n\end{array}}{\psi'}$$

Same scenario as before, but now β must be unifiable with ϕ_i, and α_1 must be unifiable with ϕ_{il}, for some l.

Apply the unifier.

We replace ϕ'_i by the premises of the rule except first premise. $\alpha'_2, \ldots, \alpha'_m$ inherit the assumptions of ϕ'_i, except ϕ'_{il}.

Destruct-Resolution

$$\frac{\begin{array}{c}[\phi_{i1} \cdots \phi_{il} \cdots \phi_{ik_i}] \\ \vdots \\ \phi_1 \quad \cdots \quad \phi_i \quad \cdots \quad \phi_n\end{array}}{\psi} \qquad \frac{\alpha}{\beta}$$

Simple rule

Destruct-Resolution

$$\frac{\begin{array}{c}\alpha \\ \hline \beta\end{array} \qquad \begin{array}{c}[\phi_{i1} \cdots \phi_{il} \cdots \phi_{ik_i}] \\ \vdots \\ \phi_1 \quad \cdots \quad \phi_i \quad \cdots \quad \phi_n\end{array}}{\psi}$$

Simple rule, and α must be unifiable with ϕ_{il}, for some l.

Destruct-Resolution

$$\frac{[\phi'_{i1} \cdots \phi'_{il} \cdots \phi'_{ik_i}] \quad \alpha'}{\beta'}$$

$$\vdots$$

$$\frac{\phi'_1 \quad \cdots \quad \phi'_i \quad \cdots \quad \phi'_n}{\psi'}$$

Simple rule, and α must be unifiable with ϕ_{il}, for some l.

We apply the unifier.

Destruct-Resolution

$$\frac{\begin{array}{c}[\phi'_{i1} \cdots \beta' \cdots \phi'_{ik_i}] \\ \vdots \\ \phi'_1 \quad \cdots \quad \phi'_i \quad \cdots \quad \phi'_n\end{array}}{\psi'}$$

Simple rule, and α must be unifiable with ϕ_{il}, for some l.

We apply the unifier.

We replace premise ϕ'_{il} with the conclusion of the rule.

Summary on Resolution

- Build proof resembling sequent style notation;
- technically: replace goals with rule premises, or goal premises with rule conclusions;
- metavariables and unification to obtain appropriate instance of rule, delay commitments;
- lifting over parameters and assumptions;
- various techniques to manipulate premises or conclusions, as convenient: `rule`, `erule`, `drule`.

More Detailed Explanations

Prolog

Prolog is a logic programming language.

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 $[\![\alpha_1; \ldots; \alpha_m]\!] \Longrightarrow \beta$). It is possible to write a little tactic program in Isabelle that "implements" a (Higher-order) Prolog interpreter.

Simple ϕ_i

ϕ_i is the selected subgoal. Isabelle kernel tactics can address with the i directly a selected subgoal. In the ISAR language, one writes:

$$\texttt{prefer i}; \texttt{apply}\ (\mathit{tactic}\ \mathit{rule})$$

With `defer i` a subgoal may be pushed towards the end of the subgoal list.

We assume here that ϕ_i is a formula, i.e., it contains no $\Longrightarrow$ (metalevel implication). The form of the other subgoals $\phi_1, \ldots, \phi_{i-1}, \phi_{i+1}, \ldots, \phi_n$ is arbitrary.

Prime ($'$)

In all illustrations that follow, we use $'$ to suggest the application of the appropriate unifier.

Metalevel Universal Quantification

$\bigwedge$ is the metalevel universal quantification (also written !!). If a goal is preceded by $\bigwedge x$, this means that Isabelle treats x as a fresh free variable (also in user definined substitutions).

Lifting over Parameters

The metavariables of the rule are made dependent on x. That is to say, each metavariable $?X$ is replaced by a $?X(x)$. You may also say that $?X$ is now a Skolem function of x.

This process is called lifting the rule over the parameter x.

Lifting over Assumptions

Each premise of the rule, as well as the conclusion of the rule, are preceded by the assumptions $[\![\phi_{i1}, \ldots, \phi_{ik_i}]\!]$ of the current subgoals. Actually, the rule

$$\dfrac{\begin{array}{c}[\phi_{i1} \cdots \phi_{ik_i}] \\ \vdots \\ \alpha_1\end{array} \quad \cdots \quad \begin{array}{c}[\phi_{i1} \cdots \phi_{ik_i}] \\ \vdots \\ \alpha_m\end{array}}{\begin{array}{c}[\phi_{i1} \cdots \phi_{ik_i}] \\ \vdots \\ \beta\end{array}}$$

may look different from any rules you have seen so far, but it can be formally derived from the rule:

$$\frac{\alpha_1 \quad \cdots \quad \alpha_m}{\beta}$$

The derived rule should be read as: If for all $j \in \{1, \ldots, m\}$, we can derive α_j from $\phi_{i1}, \ldots, \phi_{ik_i}$, then we can derive β from $\phi_{i1}, \ldots, \phi_{ik_i}$.

Unifiability

Still assuming that ϕ_i and β are unifiable.

A Trivial Unification

Both the subgoal and the conclusion of the lifted rule are preceded by assumptions $\phi_{i1}, \ldots, \phi_{ik_i}$. Hence the assumption list of the subgoal and the assumption list of the rule are trivially unifiable since they are identical.

Folding Assumptions

Generally, Isabelle makes no distinction between

$$[\![\psi_1; \ldots; \psi_n]\!] \Longrightarrow [\![\mu_1; \ldots; \mu_k]\!] \Longrightarrow \phi$$

and

$$[\![\psi_1; \ldots; \psi_n; \mu_1; \ldots; \mu_k]\!] \Longrightarrow \phi$$

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.

Same as Resolution

So the scenario looks as for resolution with lifting over assumptions. However, this time we do not show the lifting over assumptions in our animation.

The Rationale of Elimination-Resolution

Elimination-resolution is used to transform a formula in the assumption list.

For example, if the current goal is

$$\frac{\begin{array}{c}[A \wedge B] \\ \vdots \\ B\end{array}}{A \wedge B \rightarrow B}$$

and the rule is

$$\frac{P \wedge Q \qquad \begin{array}{c}[P; Q] \\ \vdots \\ R\end{array}}{R}\ \wedge\text{-}E$$

then the result of elimination resolution is

$$\frac{\begin{array}{c}[A; B] \\ \vdots \\ B\end{array}}{A \wedge B \rightarrow B}$$

Elimination resolution plays a key-role in case-distinction proofs and brings a forward proof element into backward proofs. The name of elimination resolution is motivated by the name for a particular type of rules in natural deduction calculi called elimination rules. Note that the first premise of a rule plays a distinguished role in elimination resolution.

The Rationale of Destruct-Resolution

Destruct-resolution is used to replace a formula in the assumption list by the conclusion of a rule.

For example, if the current goal is

$$\frac{\begin{array}{c}[A \wedge B] \\ \vdots \\ B\end{array}}{A \wedge B \rightarrow B}$$

and the rule is

$$\frac{P \wedge Q}{Q}\ \texttt{conjunct2}$$

then the result of destruct-resolution is

$$\frac{\begin{array}{c}[B]\\ \vdots \\ B\end{array}}{A \wedge B \rightarrow B}$$

The name of destruction resolution is motivated by the name for a particular type of rules in natural deduction calculi called destruction rules.

Isabelle: Automation by Proof Search

Burkhart Wolff

Outline of this Part

- Proof search (à la tableaux proving) and backtracking
- Making Calculi more deterministic
- Proof procedures

Proof Search and Backtracking

- Need for more automation
- Some aspects in proof construction are highly non-deterministic:
  - unification: which unifier to choose?
  - resolution: where to apply a rule (which 'subgoal')?
  - which rule to apply?
- How to organize proof-search technically?

Organizing Proof Search: Idea 1

Organize proof search as a tree of theorems (`thm`'s).

A sketch of an exemplary proof search:

Organizing Proof Search: Idea 1

Organize proof search as a tree of theorems (`thm`'s).

A sketch of an exemplary proof search:

$$\boxed{\phi \Rightarrow \phi}$$

`Goal` ϕ will create the proof state $\phi \Longrightarrow \phi$.

Organizing Proof Search: Idea 1

Organize proof search as a tree of theorems (`thm`'s).

A sketch of an exemplary proof search:

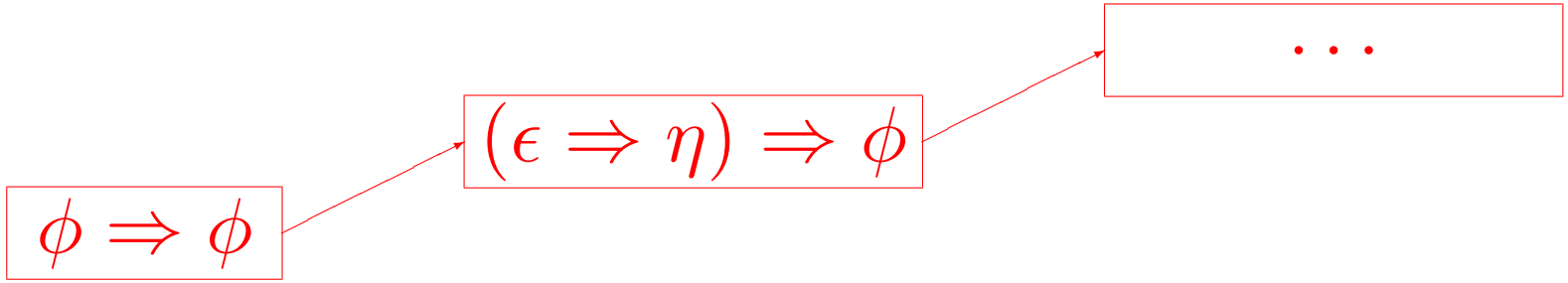
$$\boxed{\phi \Rightarrow \phi} \longrightarrow \boxed{(\epsilon \Rightarrow \eta) \Rightarrow \phi}$$

One tactic step (`apply`...) transforms into proof state $(\epsilon \Rightarrow \eta) \Rightarrow \phi$.

Organizing Proof Search: Idea 1

Organize proof search as a tree of theorems (`thm`'s).

A sketch of an exemplary proof search:

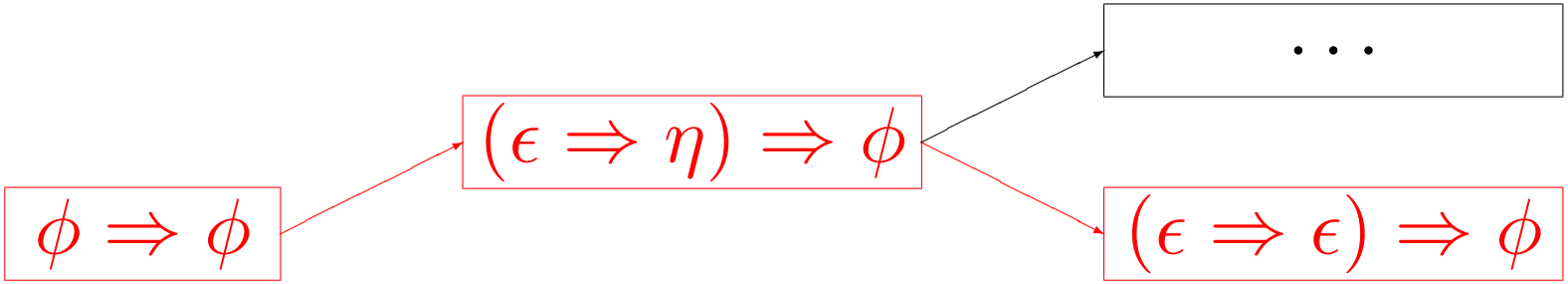


Next tactic step yields dead-end (no tactic application possible).

Organizing Proof Search: Idea 1

Organize proof search as a tree of theorems (`thm`'s).

A sketch of an exemplary proof search:

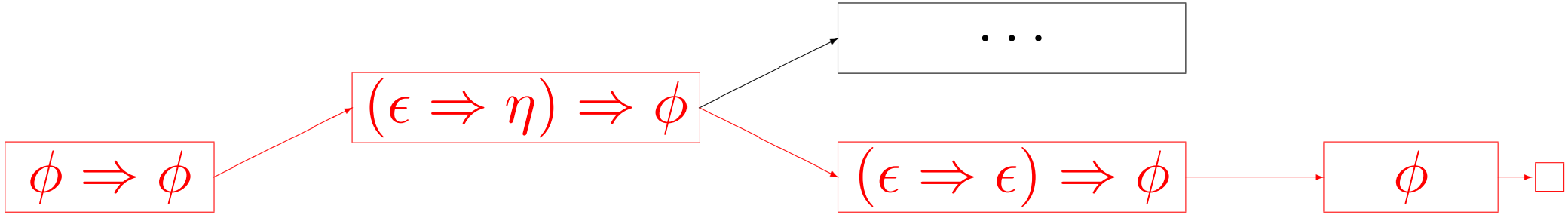


`back()`; tries an alternative successor of $(\epsilon \Rightarrow \eta) \Rightarrow \phi$.

Organizing Proof Search: Idea 1

Organize proof search as a tree of theorems (`thm`'s).

A sketch of an exemplary proof search:

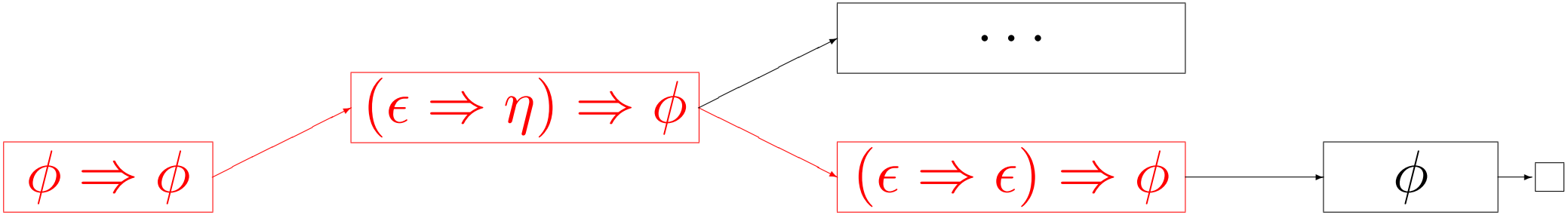


Now $(\epsilon \Rightarrow \epsilon) \Rightarrow \phi$ is solvable using `assume`/`atac`. `done`/`qed`.

Organizing Proof Search: Idea 1

Organize proof search as a tree of theorems (`thm`'s).

A sketch of an exemplary proof search:

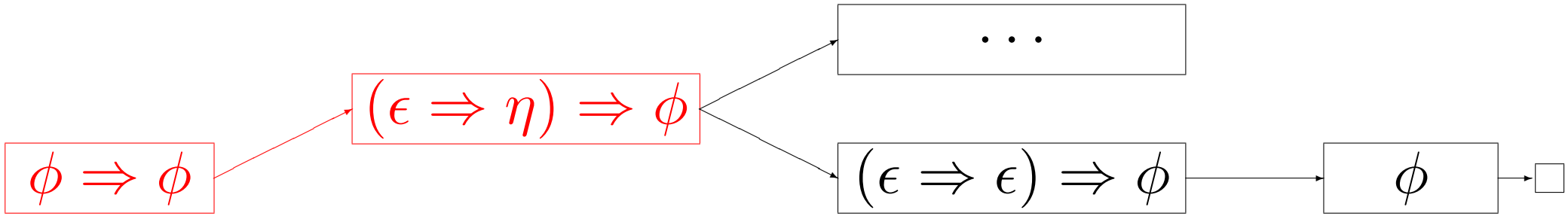


Use `undo` three times to go to previous proof states.

Organizing Proof Search: Idea 1

Organize proof search as a tree of theorems (`thm`'s).

A sketch of an exemplary proof search:

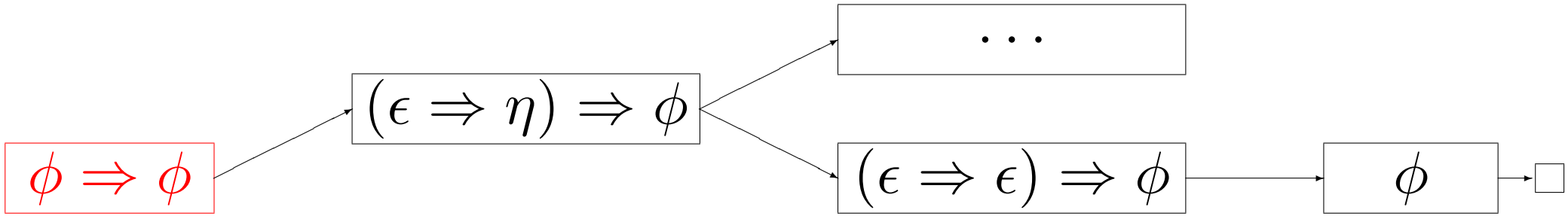


Use `undo` three times to go to previous proof states.

Organizing Proof Search: Idea 1

Organize proof search as a tree of theorems (`thm`'s).

A sketch of an exemplary proof search:

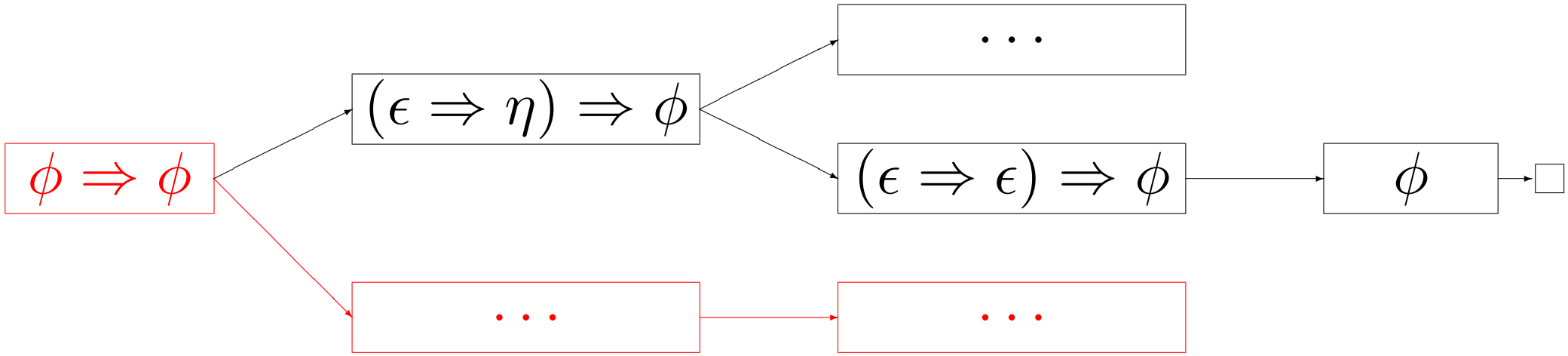


Use `undo` three times to go to previous proof states.

Organizing Proof Search: Idea 1

Organize proof search as a tree of theorems (`thm`'s).

A sketch of an exemplary proof search:

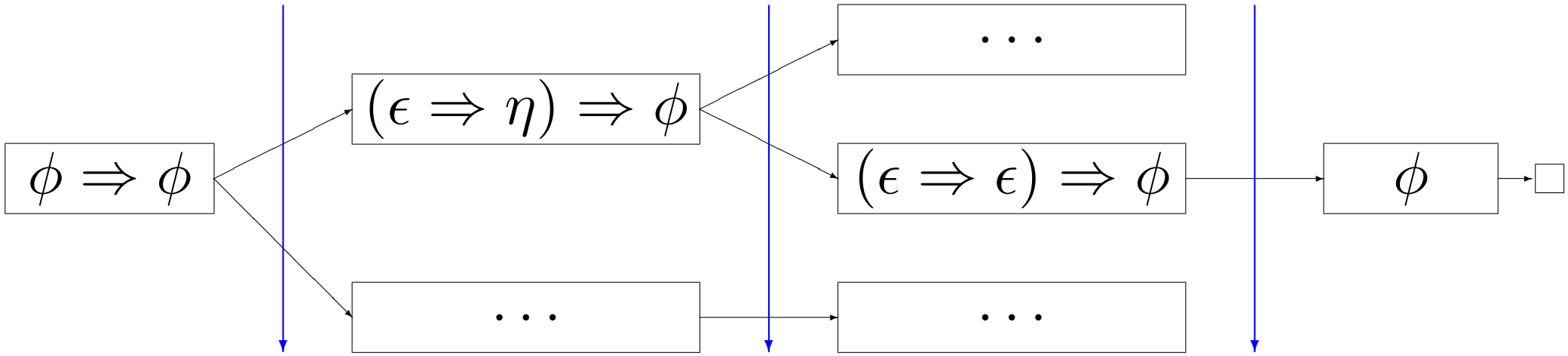


Use the `back` command to try alternative successor. . . .

Organizing Proof Search: Idea 1

Organize proof search as a tree of theorems (`thm`'s).

A sketch of an exemplary proof search:



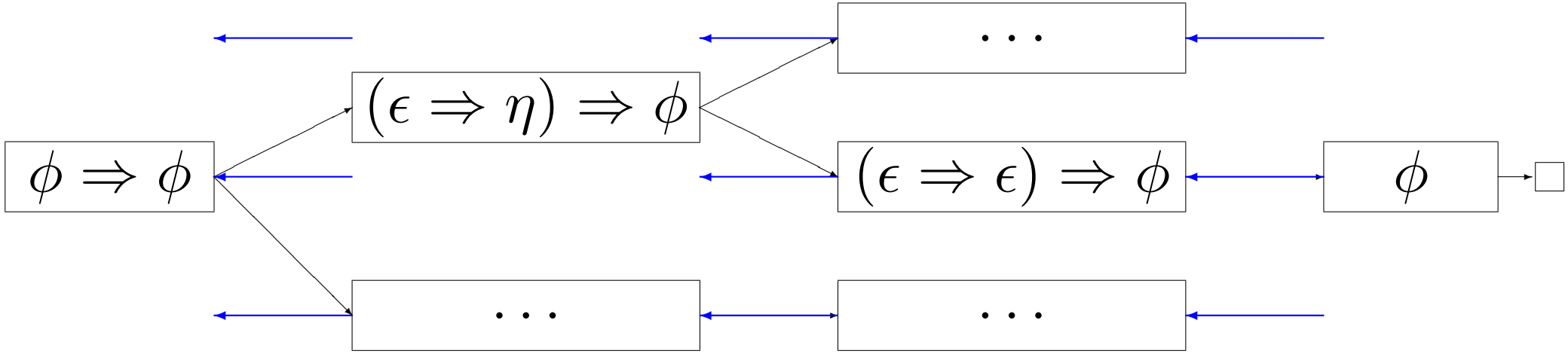
Summary:

`back` to try alternative successors ($\Rightarrow$ different unifiers).

Organizing Proof Search: Idea 1

Organize proof search as a tree of theorems (`thm`'s).

A sketch of an exemplary proof search:



Summary:

`back` to try alternative successors ($\Rightarrow$ different unifiers).

`undo` to go to previous proof state.

Problems with Idea 1

- Branching of the tree infinite in general (HO-unification)
- Explicit tree representation expensive in time and space
- Not very abstract

Organizing Proof Search: Idea 2

Organize proof search as a relation on theorems (`thm`'s)

$$prooftrees = \mathcal{P}(\texttt{thm} \times \texttt{thm})$$

Advantage: an abstract algebra

- $PT_1 \circ PT_2$: sequential composition ("then")
- $PT_1 \cup PT_2$: alternative of proof attempts ("or")
- PT^* : reflexive transitive closure ("repeat ")
- $(\phi \Rightarrow \phi, \phi) \in PT^* \quad \equiv \quad$ "there is a proof for ϕ"

Problems with Idea 2

- Union $\cup$ is difficult to implement (needs comparison with all previous results).
- More operational, strategic interpretations of union $\cup$ are desirable (try this — then that, interleave attempts in PT_1 with attempts in PT_2, and so forth).

Organizing Proof Search: Idea 3

Organize proof search as a function on theorems (`thm`'s)

$$\texttt{type tactic} = \texttt{thm} \rightarrow \texttt{thm seq}$$

where `seq` is the type constructor for infinite lists.

This allows us to have in ISAR resp. in Isabelle/ML:

- "," or `THEN`
- "|" or `ORELSE`
- "$_*$" or `REPEAT`
- only at Isabelle/ML: `INTLEAVE`, `BREADTHFIRST`, `DEPTHFIRST`, . . .

Making Calculi more Deterministic

Observation: Some rules can always be applied blindly in backward reasoning, e.g. $\rightarrow$-*I* or $\wedge$-*I*.

$$\vdash (\rho \wedge \phi) \rightarrow \psi \rightarrow \phi$$

The topmost connective is $\rightarrow$, which asks for $\rightarrow$-*I*.

Making Calculi more Deterministic

Observation: Some rules can always be applied blindly in backward reasoning, e.g. $\rightarrow$-*I* or $\wedge$-*I*.

$$\frac{\rho \wedge \phi \vdash \psi \rightarrow \phi}{\vdash (\rho \wedge \phi) \rightarrow \psi \rightarrow \phi} \rightarrow\text{-}I$$

The topmost connective is $\rightarrow$, which asks for $\rightarrow$-*I*.Again $\rightarrow$-*I*.

Making Calculi more Deterministic

Observation: Some rules can always be applied blindly in backward reasoning, e.g. $\rightarrow$-*I* or $\wedge$-*I*.

$$\dfrac{\dfrac{\rho \wedge \phi, \psi \vdash \phi}{\rho \wedge \phi \vdash \psi \rightarrow \phi}\ \rightarrow\text{-}I}{\vdash (\rho \wedge \phi) \rightarrow \psi \rightarrow \phi}\ \rightarrow\text{-}I$$

The topmost connective is $\rightarrow$, which asks for $\rightarrow$-*I*.Again $\rightarrow$-*I*.To decompose the assumption $\rho \wedge \phi$, use $\wedge$-*E*′.

Making Calculi more Deterministic

Observation: Some rules can always be applied blindly in backward reasoning, e.g. $\rightarrow$-*I* or $\wedge$-*I*.

$$\dfrac{\dfrac{\dfrac{\rho, \phi, \psi \vdash \phi}{\rho \wedge \phi, \psi \vdash \phi}\ \wedge\text{-}E'}{\rho \wedge \phi \vdash \psi \rightarrow \phi}\ \rightarrow\text{-}I}{\vdash (\rho \wedge \phi) \rightarrow \psi \rightarrow \phi}\ \rightarrow\text{-}I$$

The topmost connective is $\rightarrow$, which asks for $\rightarrow$-*I*.Again $\rightarrow$-*I*.To decompose the assumption $\rho \wedge \phi$, use $\wedge$-*E*′.The proof can be completed by assumption.

Problematic Rules

Others are problematic, e.g.:

$$\frac{\Gamma \vdash B}{A, \Gamma \vdash B}\ \textit{weaken} \qquad \frac{\Gamma \vdash B}{\Gamma \vdash A \vee B}\ \text{disjI2} \qquad \frac{\Gamma \vdash \neg\neg A}{\Gamma \vdash A}\ \text{notnotD}$$

Problematic Rules

Others are problematic, e.g.:

$$\frac{\Gamma \vdash B}{A, \Gamma \vdash B}\ \textit{weaken} \qquad \frac{\Gamma \vdash B}{\Gamma \vdash A \vee B}\ \text{disjI2} \qquad \frac{\Gamma \vdash \neg\neg A}{\Gamma \vdash A}\ \text{notnotD}$$

But: proof rules can be tailored such that they may be applied blindly.

Example: $\wedge$-E'

First approach: getting rid of critical rules by fusing them into others.

Consider:

$$\frac{A, B, \Gamma \vdash C}{A \wedge B, \Gamma \vdash C}\ \wedge\text{-}E'$$

It is instructive to reconsider the derivation of $\wedge$-E' which uses weakining inside.

The method `erule` (corresponding to `etac`) has the effect of "internalizing" weakening.

Example: $contraposXX$

Following the fusion approach, we also get alternative versions of contraposition rules:

$$\frac{B, \Gamma \vdash A}{\neg A, \Gamma \vdash \neg B}\ \text{contraposNN} \qquad \frac{\neg B, \Gamma \vdash A}{\neg A, \Gamma \vdash B}\ \text{contraposNP}$$

$$\frac{B, \Gamma \vdash \neg A}{A, \Gamma \vdash \neg B}\ \text{contraposPN} \qquad \frac{B, \Gamma \vdash A}{\neg A, \Gamma \vdash \neg B}\ \text{contraposPP}$$

Thus, with contraposNN, we incorporate the elimination of superfluous negations. contraposPN is useful but can not be applied "blindly" (non-termination).

Example: $\wedge$-E'

Second approach: Use only rules that transform the proof state equivalently (only use "safe rules" or "analytic tableaux rules").

Instead of

$$\frac{\Gamma \vdash B}{\Gamma \vdash A \vee B}\ \text{disjI2}$$

we use:

$$\frac{\neg B, \Gamma \vdash A}{\Gamma \vdash A \vee B}\ \text{disjCI}$$

which does not lose information and avoids backtracking.

Adapting Rules for Automated Proof Search

Based on disjCI and the contraposXX-rules, the following example is deterministic:

$$\frac{}{\vdash (\alpha \to \beta) \vee (\beta \to \alpha)}\ \text{disjCI1}$$

Adapting Rules for Automated Proof Search

Based on disjCI and the contraposXX-rules, the following example is deterministic:

$$\dfrac{\dfrac{}{\neg(\alpha \to \beta) \vdash \beta \to \alpha}\ {\to}\text{-}I}{\vdash (\alpha \to \beta) \vee (\beta \to \alpha)}\ \text{disjCI1}$$

Neither $\vee$-*IL* nor $\vee$-*IR* would work here. Uses classical logic.

Adapting Rules for Automated Proof Search

Based on disjCI and the contraposXX-rules, the following example is deterministic:

$$\dfrac{\dfrac{\dfrac{}{\neg(\alpha \to \beta), \beta \vdash \alpha}\,\text{contraposNP}}{\neg(\alpha \to \beta) \vdash \beta \to \alpha}\,\to\text{-}I}{\vdash (\alpha \to \beta) \vee (\beta \to \alpha)}\,\text{disjCI1}$$

Adapting Rules for Automated Proof Search

Based on disjCI and the contraposXX-rules, the following example is deterministic:

$$\dfrac{\dfrac{\dfrac{\dfrac{\neg\alpha, \alpha, \beta \vdash \beta}{\neg\alpha, \beta \vdash \alpha \to \beta}\ {\to}\text{-}I}{\neg(\alpha \to \beta), \beta \vdash \alpha}\ \text{contraposNP}}{\neg(\alpha \to \beta) \vdash \beta \to \alpha}\ {\to}\text{-}I}{\vdash (\alpha \to \beta) \vee (\beta \to \alpha)}\ \text{disjCI1}$$

Principle: Emulate sequent calculus with derived rules.

The safe, but non-terminating contraposNP can be avoided by fusing it with all logical junctors.(In this case: $\to$).

Handling Quantifiers

Can derive $\forall$-E' ($\equiv$ `allE`) using $\forall$-E ($\equiv$ `spec`):

$$\dfrac{\forall x.A(x) \qquad \begin{array}{c}[A(x) \qquad\qquad] \\ \vdots \\ B\end{array}}{B}\ \forall\text{-}E'$$

What is the difference to $\exists$-E?

Handling Quantifiers

Can derive $\forall$-E' ($\equiv$ `allE`) using $\forall$-E ($\equiv$ `spec`):

$$\dfrac{\forall x.A(x) \qquad \begin{array}{c}[A(x) \qquad\qquad] \\ \vdots \\ B\end{array}}{B}\ \forall\text{-}E'$$

What is the difference to $\exists$-E?

Problem: $\forall x.A(x)$ may still be needed.

Handling Quantifiers

Can derive $\forall$-E' ($\equiv$ `allE`) using $\forall$-E ($\equiv$ `spec`):

$$\dfrac{\forall x.A(x) \qquad \begin{array}{c}[A(x), \forall x.A(x)] \\ \vdots \\ B\end{array}}{B}\ \forall\text{-}dupE$$

What is the difference to $\exists$-E?

Problem: $\forall x.A(x)$ may still be needed.

Principle: Introduce duplicating rules. Turns search infinite!

Check out `allE` and `all_dupE` in `IFOL`!

Proof Procedures (Simplified)

Proof Procedures (Simplified)

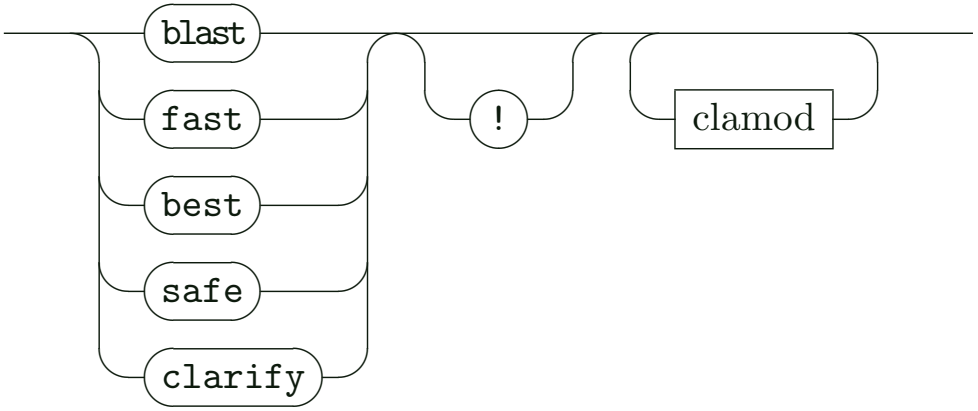
Tactics in Isabelle are performed in order:

1. DEPTHSOLVE(
   REPEAT(rtac $safe_I_rules$ ORELSE etac $safe_E_rules$))
2. canonize: propagate "$x = t$" throughout subgoal
3. rtac $unsafe_I_rules$ ORELSE etac $unsafe_E_rules$
4. atac

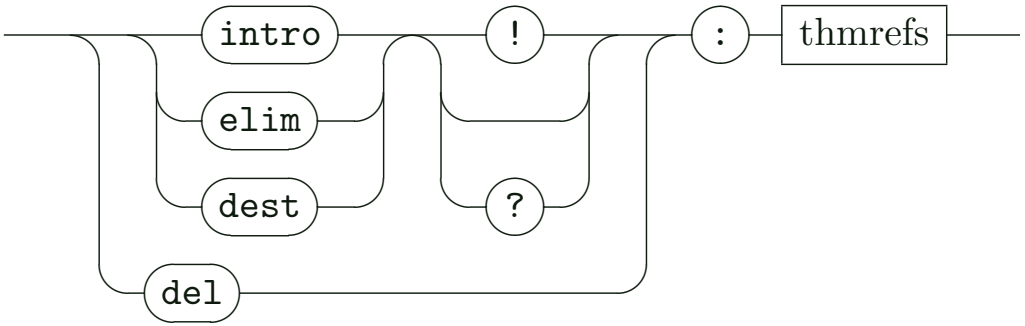
In ISAR, rtac is rule,etac is erule, . . .

Combined Proof Search Tactics in ISAR

On the ISAR-level, the syntax for commands accessing the "provers" looks as follows:



clamod



`clamod` allows for introducing new rules (`thm`'s) as introduction, elimination or destruction rules. Rules classified with bang "!" were applied earlier and more agressively as "safe rules".

These commands were mapped to the SML-tactics (described in more detail in the Isabelle Reference Manual [Pau03]).

Safe and Unsafe Rules

On the Isabelle SML level, the rules and their classification were maintained in the data structure `claset`, and accessed by functions of type `claset` $*$ `thm list` $\longrightarrow$ `claset`.

Class:	To add use function:
Safe introduction rules	`addSIs`
Safe elimination rules	`addSEs`
Unsafe introduction rules	`addIs`
Unsafe elimination rules	`addEs`

Combined Proof Search Tactics

- `fast_tac` : `claset` $\longrightarrow$ `int` $\longrightarrow$ `tactic`
  (safe and unsafe steps in depth-first stategy)
- `best_tac` : `claset` $\longrightarrow$ `int` $\longrightarrow$ `tactic`
  (safe and unsafe steps in breadth-first stategy)
- `blast_tac` : `claset` $\longrightarrow$ `int` $\longrightarrow$ `tactic`
  (like `fast_tac`, but often more powerful)

More details can be found in the Isabelle Reference Manual[Pau03].

Summary on Automated Proof Search

- Proof search can be organized as a tree of theorems.
- Calculi can be set up to facilitate proof search (although this must be done by specialists).
- Combined with search strategies, powerful automatic procedures arise. Can prove well-known hard problems such as $((\exists y.\forall x.J(y,x) \vee \neg J(x,x)) \rightarrow \neg(\forall x.\exists y.\forall z.J(z,y) \vee \neg J(z,x))$
- Unfortunately, failure is difficult to interpret.

More Detailed Explanations

Notion

In this lecture we use both, the ISAR synatx and the "classical" ML based syntax of Isabelle. We first denote the ISAR syntax, followed by the ML syntax, e.g. `assume`/`atac`.

Need for Automation

We have seen in the exercises that proving on a stepwise basis is very tedious and yearns for automation.

Efficiency considerations are also important for automation. The non-determinacy in proof search may lead to deep backtracking which should therefore be avoided.

Idea 1: A Tree of Theorems

We have seen in the previous lecture that resolution transforms a proof state into a new proof state. Since in general, a proof state has several successor states (states that can be obtained by one resolution step), conceptually one obtains a tree where the children of a state are the successors.

The essential point of idea 1 is that the tree is constructed explicitly, as a data-structure.

$\phi \Longrightarrow \phi$?

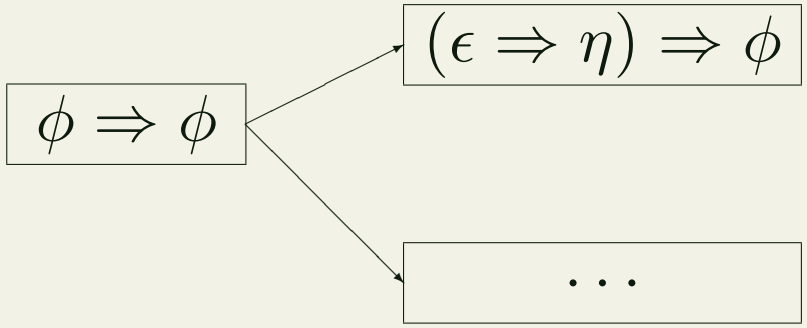
The initial proof state is $\phi \Longrightarrow \phi$. Isabelle will display this as

```
Level 1 : (1 subgoal)
```
ϕ
```
 1.
```
ϕ

Technically, the proof state is an Isabelle theorem (`thm`), i.e. something which Isabelle considers as proven. The aim of a proof search in backward proof is to transform $\phi \Longrightarrow \phi$ into ϕ (ϕ can be shown if I assume nothing).

Idea 2: A Relation on Theorems

One can look at a fragment of a tree of theorems as in idea 1, e.g.:



One could say that each tactic application (with a particular rule) gives rise to a relations on theorems. That is to say, ϕ and ϕ' are in the relation if ϕ' is a successor proof state of ϕ.

This is abstract in that there is no order among the successors of a proof state.

Also, one does not represent a tree explicitly.

Sequential Composition

Given two relations between `thm`'s, PT_1 and PT_2, we define $PT_1 \circ PT_2$ as the relation

$$\{(\phi, \psi) \mid \text{there is } \eta \text{ such that } (\phi, \eta) \in PT_1 \text{ and } (\eta, \psi) \in PT_2\}$$

Union of Relations

The union of two relations is defined as usual for sets. If PT_1 and PT_2 each model the application of a particular tactic, then $PT_1 \cup PT_2$ models the application of "first tactic or second tactic".

Reflexive Transitive Closure

PT^* is inductively defined as the smallest set where

- $(\phi, \phi) \in PT^*$ for all ϕ;
- if $(\phi, \eta) \in PT$ and $(\eta, \psi) \in PT^*$ then $(\phi, \psi) \in PT^*$.

So if PT models the application of a particular tactic, then PT^* models the application of that tactic arbitrarily many times

Idea 2: A Function on Theorems

Idea 3 differs from idea 2 in that it is less abstract, more 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, but by higher-order functions that can compute the rest of a sequence step by step.

Infinite Lists

For any type τ, the type τ seq (recall the notation) is the type of (possibly) infinite lists of elements of type τ. This is of course an abstract datatype. There should be functions to return the head and the tail of such an infinite list.

An abstract datatype is a type whose terms cannot be represented explicitly and accessed directly, but only via certain functions for that type.

Tacticals

- THEN
- ORELSE
- REPEAT
- INTLEAVE, BREADTHFIRST, DEPTHFIRST, ...

are called tacticals.

Tacticals are operations on tactics. They play an important role in automating proofs in Isabelle. The most basic tacticals are THEN and ORELSE. Both of those tacticals are of type tactic $*$ tactic $\longrightarrow$ 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 [Pau03, Ch. 4].

$\wedge$-E

In Isabelle notation, it looks as follows:

$$[\![P \wedge Q;\ [\![P;\ Q]\!] \Longrightarrow R]\!] \Longrightarrow R$$

claset

`claset` is an abstract datatype. Overloading notation, `claset` is also an ML unit function which will return a term of that datatype when applied to (), namely, the current classifier set.

A classifier set determines which rules are safe and unsafe introduction, respectively elimination rules. The current classifier set is a classifier set used by default in certain tactics.

The current classifier set can be accessed via special functions for that purpose.

Accessing the `claset`

The functions `addSIs`, `addSEs`, `addIs`, `addEs` are all of type `claset` $*$ `thm list` $\longrightarrow$ `claset`. They add rules to the current classifier set. For example, `addSIs` adds a rule as safe introduction rule.

Emulating the Sequent Calculus

The sequent calculus works with expressions of the form $A_1, \ldots, A_n \vdash B_1, \ldots, B_m$ which should be interpreted as: under the assumptions $A_1, \ldots, A_n$, at least one of $B_1, \ldots, B_m$ can be proven. So as a formula, this would be $A_1 \wedge \ldots \wedge A_n \rightarrow B_1 \vee \ldots \vee B_m$.

In Isabelle (and the proof trees we have seen, e.g,. in this lecture), we only have sequents with one formula to the right of the $\vdash$. We have said that we use sequent notation.

Deriving `allE`

You should do it in Isabelle. The rule is:

$$[\![\texttt{ALL}\ x.\ P(x);\ P(x) \Longrightarrow R]\!] \Longrightarrow R$$

The Rule $\vee$-*swap*

The rule `disjCI` is

$$\frac{\neg A, \Gamma \vdash B}{\Gamma \vdash A \vee B}\ \texttt{disjCI}$$

To derive it you need classical reasoning, as the rule exploits the equivalence of A and $\neg\neg A$ (then the rule follows immediately from $\rightarrow$-*I*).

The Rule impE

The rule impE is

$$\frac{A, \neg C, \Gamma \vdash B}{\neg(A \to B), \Gamma \vdash C}\ \text{impE}$$

It essentially "fuses" `contraposNP`, which can not be applied "blindly" due to non-termination, with $\to$-*I*.

This is a standard technique in Isabelle called swapping. In generally, if 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 [Pau03, Section 11.2].

Duplicating Rules

You should recall that elimination rules are used in combination with `erule`/`etac`. Using `allE` will eliminate the quantifier.

You should try a proof of the formula $(\forall x.P(x)) \rightarrow (P(a) \wedge P(b))$ in Isabelle to convince yourself that this is a problem since the quantified formula $\forall x.P(x)$ is needed twice as an assumption, with two different instantiations of x.

The duplicating rule $\forall$-*dupE* has the effect that the universally quantified formula will still remain as an assumption.

Proof Procedures

Tactics in Isabelle are performed in order:

1. DEPTHSOLVE(REPEAT(rtac *safe_I_rules* ORELSE etac *safe_E_rules*));
2. canonize: propagate "$x = t$" . . . throughout subgoal;
3. rtac *unsafe_I_rules* ORELSE etac *unsafe_E_rules*;
4. atac.

One elementary proof step consists of trying a safe introduction rule with rtac, or, if that is not possible, a safe elimination rule with etac. This will be repeated as long as possible.

Then in the current subgoal, any assumption of the form $x = t$ (where x is a metavariable) will be propagated throughout the subgoal, i.e., all occurrences of x wil be replaced by t.

Then Isabelle will try one application of an unsafe introduction rule with `rtac`, or, if that is not possible, an unsafe elimination rule with `etac`. Finally, she will use `assumption/atac`. Note that `assumption/atac` is unsafe. In general, there are several premises in a subgoal and `atac` may unify the conclusion of the subgoal with the wrong premise. Different search strategies were applied.

Isabelle: Term Rewriting

Burkhart Wolff

Outline of this Part

- Higher-order rewriting
- Extensions: Ordered, pattern, congruence, splitting rewriting
- Organizing simplification rules

In this context, a term is a λ-term, since we use the λ-calculus to encode object logics.

Higher-Order Rewriting

Motivation:

- Simplification is a very important part of deduction, e.g.:

$$\begin{aligned} 0 + (x + 0) &= x \\ [a, b, d] \mathbin{@} [a, b] &= [a, b, d, a, b] \end{aligned}$$

- Based on rewrite rules as in functional programming:

$$\begin{aligned} x + 0 &= x, & 0 + x &= x \\ [] \mathbin{@} X &= X, & (x :: X) \mathbin{@} Y &= x :: (X \mathbin{@} Y) \end{aligned}$$

Term Rewriting: Foundation

- Recall: An equational theory consists of rules

$$\frac{}{x = x}\,\textit{refl} \qquad \frac{x = y}{y = x}\,\textit{sym} \qquad \frac{x = y \quad y = z}{x = z}\,\textit{trans}$$

$$\frac{x = y \quad P(x)}{P(y)}\,\textit{subst}$$

- 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, i.e. they are applied from left to right.

Algorithm $simplify_R$

- We assume a rule set R
- An equation is solved if it has the form $e = e$

Algorithm $simplify_R$

- We assume a rule set R
- An equation is solved if it has the form $e = e$
- An equation is simplified by:

  $simplify_R(e = e') =>$

  $repeat$

  (a) pick terms h and t such that $(e = e') \equiv h(t)$

Algorithm $simplify_R$

- We assume a rule set R
- An equation is solved if it has the form $e = e$
- An equation is simplified by:

  $simplify_R(e = e') =>$

  $repeat$

  (a) pick terms h and t such that $(e = e') \equiv h(t)$

  (b) pick a rewrite rule $\phi_1 = \psi_1, \ldots, \phi_n = \psi_n \Longrightarrow \phi = \psi$ from R, match (unify) ϕ against t, i.e., find θ such that $\phi\theta = t$

Algorithm $simplify_R$

- We assume a rule set R
- An equation is solved if it has the form $e = e$
- An equation is simplified by:

  $simplify_R(e = e') =>$

  $repeat$

  (a) pick terms h and t such that $(e = e') \equiv h(t)$

  (b) pick a rewrite rule $\phi_1 = \psi_1, \ldots, \phi_n = \psi_n \Longrightarrow \phi = \psi$ from R, match (unify) ϕ against t, i.e., find θ such that $\phi\theta = t$

  (c) replace $e = e'$ by $h(\psi\theta)$ provided all $simplify((\phi_i = \psi_i)\theta)$ are solved for all $i \in \{1..n\}$

Algorithm $simplify_R$

- We assume a rule set R
- An equation is solved if it has the form $e = e$
- An equation is simplified by:

  $simplify_R(e = e') =>$

  $repeat$

  (a) pick terms h and t such that $(e = e') \equiv h(t)$

  (b) pick a rewrite rule $\phi_1 = \psi_1, \ldots, \phi_n = \psi_n \Longrightarrow \phi = \psi$ from R, match (unify) ϕ against t, i.e., find θ such that $\phi\theta = t$

  (c) replace $e = e'$ by $h(\psi\theta)$ provided all $simplify((\phi_i = \psi_i)\theta)$ are solved for all $i \in \{1..n\}$

  $until$ no replacement possible, return current $e = e'$

Algorithm $simplify_R$

- We assume a rule set R
- An equation is solved if it has the form $e = e$
- An equation is simplified by:

  $simplify_R(e = e') =>$

  $repeat$

  (a) pick terms h and t such that $(e = e') \equiv h(t)$

  (b) pick a rewrite rule $\phi_1 = \psi_1, \ldots, \phi_n = \psi_n \Longrightarrow \phi = \psi$ from R, match (unify) ϕ against t, i.e., find θ such that $\phi\theta = t$

  (c) replace $e = e'$ by $h(\psi\theta)$ provided all $simplify((\phi_i = \psi_i)\theta)$ are solved for all $i \in \{1..n\}$

  $until$ no replacement possible, return current $e = e'$

Problems with $simplify$

- This algorithm may fail because:
  - it diverges (the rules are not terminating), e.g. $x + y = y + x$ or $x = y \Longrightarrow x = y$;

Problems with $simplify$

- This algorithm may fail because:
  - it diverges (the rules are not terminating), e.g. $x + y = y + x$ or $x = y \Longrightarrow x = y$;
  - rewriting does not yield a unique normal form (the rules are not confluent), e.g. rules $a = b$, $a = c$.

Problems with $simplify$

- This algorithm may fail because:
  - it diverges (the rules are not terminating), e.g. $x + y = y + x$ or $x = y \Longrightarrow x = y$;
  - rewriting does not yield a unique normal form (the rules are not confluent), e.g. rules $a = b$, $a = c$.
- Providing criteria for terminating and confluent rule sets R is an active research area (see [BN98, Klo93], RTA, . . .).

Extensions of Rewriting

- Symmetric rules are problematic, e.g. ACI:

$$
\begin{aligned}
(x+y)+z &= x+(y+z) && \text{(A)}\\
x+y &= y+x && \text{(C)}\\
x+x &= x && \text{(I)}
\end{aligned}
$$

Extensions of Rewriting

- Symmetric rules are problematic, e.g. ACI:

$$
\begin{array}{rcll}
(x+y)+z & = & x+(y+z) & \text{(A)} \\
x+y & = & y+x & \text{(C)} \\
x+x & = & x & \text{(I)}
\end{array}
$$

- Idea: apply only if replaced term gets smaller w.r.t. some term ordering. In example, if $y+x\theta$ is smaller than $x+y\theta$.
- Ordered rewriting solves rewriting modulo ACI, using derived rules (exercise).

Extension: HO-Pattern Rewriting

Rules such as $F(G\,c) = \ldots$ lead to highly ambiguous matching and hence inefficiency.

Solution: restrict l.h.s. of a rule to higher-order patterns.

Extension: HO-Pattern Rewriting

Rules such as $F(G\,c) = \ldots$ lead to highly ambiguous matching and hence inefficiency.

Solution: restrict l.h.s. of a rule to higher-order patterns.

A term t is a HO-pattern if

- it is in β-normal form; and
- any free F in t occurs in a subterm $F\,x_1 \ldots x_n$ where the x_i are η-equivalent to distinct bound variables.

Extension: HO-Pattern Rewriting

Rules such as $F(G\,c) = \ldots$ lead to highly ambiguous matching and hence inefficiency.

Solution: restrict l.h.s. of a rule to higher-order patterns.

A term t is a HO-pattern if

- it is in β-normal form; and
- any free F in t occurs in a subterm $F\,x_1 \ldots x_n$ where the x_i are η-equivalent to distinct bound variables.

Matching (unification) is decidable, unitary ('unique') and efficient algorithms exist.

HO-Pattern Rewriting (Cont.)

A rule $\ldots \Rightarrow \phi = \psi$ is a HO-pattern rule if:

- the left-hand side ϕ is a HO-pattern;
- all free variables in ψ occur also in ϕ; and
- ϕ is constant-head, i.e. of the form $\lambda x_1..x_m.c\,p_1 \ldots p_n$ (where c is a constant, $m \geq 0$, $n \geq 0$).

HO-Pattern Rewriting (Cont.)

A rule $\ldots \Rightarrow \phi = \psi$ is a HO-pattern rule if:

- the left-hand side ϕ is a HO-pattern;
- all free variables in ψ occur also in ϕ; and
- ϕ is constant-head, i.e. of the form $\lambda x_1..x_m.c\, p_1 \ldots p_n$ (where c is a constant, $m \geq 0$, $n \geq 0$).

Example: $(\forall x.Px \wedge Qx) = (\forall x.Px) \wedge (\forall x.Qx)$

Result: HO-pattern allows for very effective quantifier reasoning.

Extension: Congruence Rewriting

Problem :

$$\texttt{if}\, A\, \texttt{then}\, P\, \texttt{else}\, Q \;=\; \texttt{if}\, A\, \texttt{then}\, P'\, \texttt{else}\, Q$$
$$\text{where } P = P' \text{ under condition } A$$

is not a rule.

Solution in Isabelle: explicitely admit this extra class of rules (congruence rules)

$$[\![A \Longrightarrow P = P']\!] \Longrightarrow$$
$$\texttt{if}\, A\, \texttt{then}\, P\, \texttt{else}\, Q \;=\; \texttt{if}\, A\, \texttt{then}\, P'\, \texttt{else}\, Q$$

Extension: Splitting Rewriting

Problem:

$$P(\texttt{if}\, A\, \texttt{then}\, x\, \texttt{else}\, y) = ((A \Longrightarrow P\, x) \wedge (\neg A \Longrightarrow P\, y))$$

is not a HO-pattern rule (since it is not constant-head).
Similar problems arise in connection with data types and their resulting case match statements (to be discussed later).
Solution in Isabelle: explicitely admit this extra class of (splitting rules).

Organizing Simplification Rules

- Standard (HO-pattern conditional ordered rewrite) rules;
- congruence rules;
- splitting rules.

In the Isabelle kernel, on the SML level, the data structure `simpset` is provided. Some operations:

- $\texttt{addsimps} : \texttt{simpset} * \texttt{thm list} \longrightarrow \texttt{simpset}$
- $\texttt{delsimps} : \texttt{simpset} * \texttt{thm list} \longrightarrow \texttt{simpset}$
- $\texttt{addcongs} : \texttt{simpset} * \texttt{thm list} \longrightarrow \texttt{simpset}$
- $\texttt{addsplits} : \texttt{simpset} * \texttt{thm list} \longrightarrow \texttt{simpset}$

Commutativity can be added without losing termination.

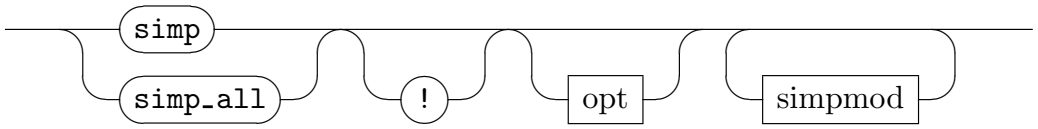
How to Apply the Simplifier?

Several versions of the simplifier in the Isabelle engine (ML-level):

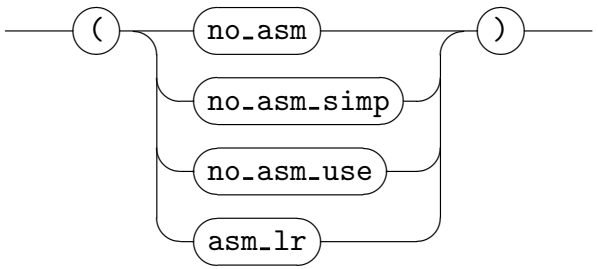
- `simp_tac` : `simpset` $\longrightarrow$ `int` $\longrightarrow$ `tactic`
- `asm_simp_tac` : `simpset` $\longrightarrow$ `int` $\longrightarrow$ `tactic`
  (includes assumptions into `simpset`)
- `asm_full_simp_tac` : `simpset` $\longrightarrow$ `int` $\longrightarrow$ `tactic`
  (rewrites assumptions, and includes them into simpset)

How to Apply the Simplifier?

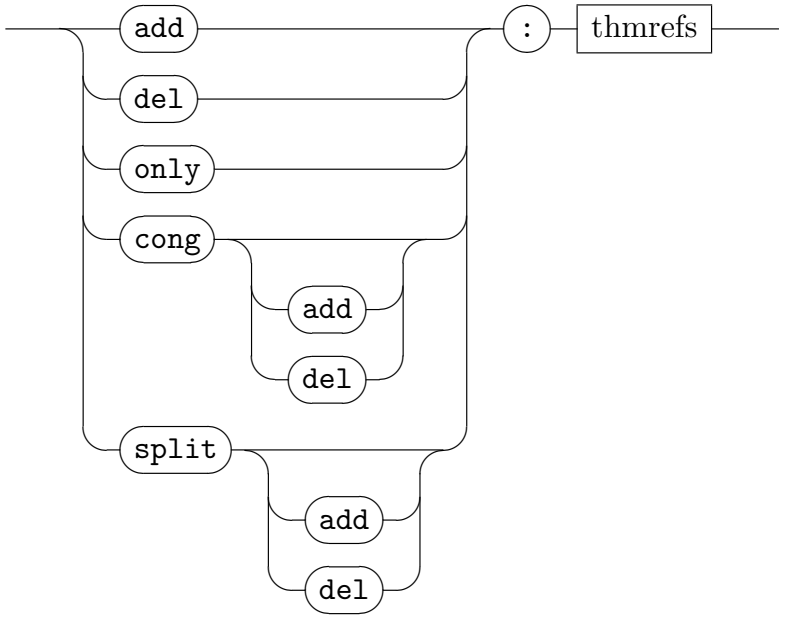
On the ISAR level, these tactics are accessible as ISAR methods and have the following syntax:



opt



simpmod



Summary on the Simplifier and Term Rewriting

Simplifier is a powerful proof tool for

- conditional equational formulas
- ACI-rewriting
- quantifier reasoning
- congruence rules
- automatic proofs by case split rules

Fortunately, failure is quite easy to interpret since even intermediate results were computed and the solving process can be traced.

Summary on Last Three Sections

- Although Isabelle is an interactive proof construction, it is a flexible environment with powerful automated proof procedures.
- For classical logic and set theory, tableau-like procedures like `blast_tac` and `fast_tac` decide many tautologies.
- For equational theories (datatypes, evaluating functional programs, but also higher-order logic) `simp_tac` decides many tautologies (and is fairly easy to control).

More Detailed Explanations

$$0 + (x + 0) = x$$

Simplifying $0 + (x + 0)$ to x is something you have learned in school. It is justified by the usual semantics of arithmetic expressions. Here, however, we want to see more formally how such simplification works, rather than why it is justified.

Lists

Lists are a common datatype in functional programming. $[a, b, d, a, b]$ is a list. Actually, this notation is syntactic sugar for $a :: (b :: (d :: (a :: (b :: [])))))$. Here, $[]$ is the empty list and $::$ is a term constructor taking an alement and a list and returning a list. @ stands for list concatenation.

Intuitively, it is clear that $[a, b, d]$ concatenated with $[a, b]$ yields $[a, b, d, a, b]$.

Term constructor is usual terminology in functional programming. In first-order logic, we would speak of a function symbol. In the λ-calculus, we would speak of a (special kind of) constant (this will become clear later).

Functional Programming

For example, the lines

$$\begin{array}{rcl} [] \mathbin{@} X & = & X \\ (x :: X) \mathbin{@} Y & = & x :: (X \mathbin{@} Y) \end{array}$$

define the list concatenation function @.

Rewrite Rules

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.

Matching

Given two terms s and t, a unifier is a substitution θ such that $s\theta = t\theta$. A match is a substitution which only instantiates one of s or t, so $s\theta = t$ or $s = t\theta$ (one should usually clarify in the given context which of the terms is instantiated).

$a = b$, $a = c$

For a rewriting system consisting of rules $a = b$, $a = c$, one cannot rewrite $b = c$ to prove the equality, although it holds:

$$\dfrac{\dfrac{a = b}{b = a}\,\textit{sym} \qquad a = c}{b = c}\,\textit{trans}$$

Term Ordering

The biggest problem for term rewriting is (non-)termination. For some crucial rules, this problem is solved by ordered term rewriting. A term ordering is any partial order between ground (i.e., not containing free variables) terms.

One can define a term ordering by giving some function, called norm, from ground terms to natural numbers. Then a term is smaller than another term if the number assigned to the first term is smaller that the number assigned to the second term.

How Ordered Rewriting Solves ACI

Consider an equational theory consisting only of those rules (apart from *refl*, *sym*, *trans*, *subst*). Apart from that, the language may contain arbitrary other constant symbols. For such a language, it is possible to give a term ordering that will assign more weight to the same term on the left-hand-side of a $+$ than on the right-hand side. We can base such a term ordering on a norm. For example, the inductive definition of a norm $|_|$ might include the line:

$$|s + t| := 2|s| + |t|$$

This means that if $|s| > |t|$, then $|s + t| = 2|s| + |t| > 2|t| + |s| = |t + s|$. This has two effects:

- Applications of (A) or (I) always decrease the weight of a term

(provided the weight of s is > 0):

$$|(s+t)+r| = 2|s+t| + |r| = 4|s| + 2|t| + |r| \quad > \\ 2|s| + 2|t| + |r| = 2|s| + |t+r| = |s+(t+r)|.$$

- Applications of (C) are only possible if the left-hand side is heavier than the right-hand side.

Isabelle internally provides a term order, and the simplifier will use this in order to avoid non-termination for commutativity and similar rules.

Now, the question arises how ACI normal forms can be computed if commutativity is now longer a problem. The problem is that commutativity and idempotence patterns overlap and for the overlapping cases:

x + (x + y) = x + y
y + (x + z) = x + y + z

own rules must be derived. By Isabelle convention, these finitely many rules were stored in own rule sets such as Un_ac which can be accessed in ISAR.

Ambiguous Matching

For higher-order rewriting, it is very problematic to have rules containing terms of the form $F(G\,c)$ on the left-hand side, where F and G are free variables and c is a constant or bound variable. The reason can be seen in an example: Suppose you want to rewrite the term $f(g(h(i\,c)))$ where f, g, h, i are all constants. There are four unifiers of $F\,(G\,c)$ and $f(g(h(i\,c)))$:

$$\begin{array}{l}\{f/F,\ (\lambda x.g(h(i\,x)))/G\},\\ \{(\lambda x.f(g\,x))/F,\ (\lambda x.h(i\,x))/G\},\\ \{(\lambda x.f(g(h\,x)))/F,\ (\lambda x.i\,x)/G\},\\ \{(\lambda x.f(g(h(i\,x))))/F,\ (\lambda x.x)/G\}.\end{array}$$

$\forall, \exists$ is a Constant

Further examples:

- $(\exists x.Px \vee Qx) = (\exists x.Px) \vee (\exists x.Qx)$
- $(\exists x.P \rightarrow Qx) = P \rightarrow (\exists x.Qx)$
- $(\exists x.Px \rightarrow Q) = (\forall x.Px) \rightarrow Q$

In these examples, you may assume that first-order logic is our object logic.

On the metalevel, and hence also for the sake of term rewriting, $\forall, \exists$ are constants.

In the notation $(\forall x.Px \wedge Qx)$, the symbols P and Q are variables.

The principle was explained thoroughly before.

Isabelle's Metalogic and Proof Objects

Burkhart Wolff

Overview

This chapter reconsiders again Isabelle as a logical framework. This involves:

Overview

This chapter reconsiders again Isabelle as a logical framework. This involves:

- its version of a typed λ-calculus

Overview

This chapter reconsiders again Isabelle as a logical framework. This involves:

- its version of a typed λ-calculus
- its elementary logic called Pure

Overview

This chapter reconsiders again Isabelle as a logical framework. This involves:

- its version of a typed λ-calculus
- its elementary logic called Pure
- a deeper understanding of *rule*/rtac etc,

Overview

This chapter reconsiders again Isabelle as a logical framework. This involves:

- its version of a typed λ-calculus
- its elementary logic called Pure
- a deeper understanding of *rule*/rtac etc,
- proof objects

Overview

This chapter reconsiders again Isabelle as a logical framework. This involves:

- its version of a typed λ-calculus
- its elementary logic called Pure
- a deeper understanding of *rule*/rtac etc,
- proof objects
- consequences

An Extension of the Typed λ-Calculus

Universal representation for object logics in Isabelle: A Typed λ-calculus extended by (parametric) polymorphism and type classes.

An Extension of the Typed λ-Calculus

Universal representation for object logics in Isabelle: A Typed λ-calculus extended by (parametric) polymorphism and type classes.

Historically, polymorphism in logics — although already used in the principia mathematica on the meta-level — is a fairly recent discovery (around 1975, first implementation: Edinburgh LCF). The consequences for Conservative Definitions have been sorted out in the early 80ies.

Polymorphism: Intuition

As in functional programming, the function $_ = _$ should be available on any type. This can be expressed by giving $_ = _$ the type $[\alpha, \alpha] \Rightarrow bool$ with α an explicit type variable as part of the type expression language.

Polymorphism: Intuition

As in functional programming, the function $_ = _$ should be available on any type. This can be expressed by giving $_ = _$ the type $[\alpha, \alpha] \Rightarrow bool$ with α an explicit type variable as part of the type expression language.

Adding type classes ("sorts of types") helps to separate universes of types from each other. $[\alpha :: term, \alpha] \Rightarrow bool$, for example, can be used to express that α may range over all types with individuals, but not predicates (i.e. $bool$ as in FOL).

Polymorphism: Intuition

As in functional programming, the function $_ = _$ should be available on any type. This can be expressed by giving $_ = _$ the type $[\alpha, \alpha] \Rightarrow bool$ with α an explicit type variable as part of the type expression language.

Adding type classes ("sorts of types") helps to separate universes of types from each other. $[\alpha :: term, \alpha] \Rightarrow bool$, for example, can be used to express that α may range over all types with individuals, but not predicates (i.e. $bool$ as in FOL).

Adding type constructors allows the introduction of $bool$, but also concepts such as $\alpha\, set$.

Polymorphism: Intuition

As in functional programming, the function $_ = _$ should be available on any type. This can be expressed by giving $_ = _$ the type $[\alpha, \alpha] \Rightarrow bool$ with α an explicit type variable as part of the type expression language.

Adding type classes ("sorts of types") helps to separate universes of types from each other. $[\alpha :: term, \alpha] \Rightarrow bool$, for example, can be used to express that α may range over all types with individuals, but not predicates (i.e. $bool$ as in FOL).

Adding type constructors allows the introduction of $bool$, but also concepts such as $\alpha\, set$.

We present a simplification of [NP95]. More formally, we have:

Syntax: Classes, Types, and Terms

Type classes (exemplary)

$$\kappa \quad ::= \quad ord \mid order \mid lattice \mid \ldots$$

Type constructors (exemplary)

$$\chi \quad ::= \quad bool \mid _ \rightarrow _ \mid ind \mid _\, list \mid _\, set \ldots$$

Polymorphic types

$$\tau \quad ::= \quad \alpha :: \{\kappa, \ldots, \kappa\} \mid (\tau, .., \tau)\chi \qquad (\alpha \text{ is type variable})$$

Raw terms (as before)

$$e \quad ::= \quad x \mid ?x \mid c \mid (ee) \mid (\lambda x^{\tau}.\, e)$$

ClaPolymorphic Type Inferences (1)

Prerequisites:

- a partial order $\leq$ on classes,
- . . . implying an equivalence on type class sets,
- a constant environment Σ, a variable environment Γ and a type environment ξ assigning to type variables (finite) sets of type classes,
- a type instance relation Δ assigning $(\kappa..\kappa)\chi$ to κ
- Type instances (denoted Θ) extend type environments to substitutions of types in terms,
- and two judgements $\Sigma, \xi \vdash \tau : \{\kappa..\kappa\}$ and $\Sigma, \Gamma, \xi \vdash e : \tau$

Polymorphic Type Inferences (2)

$$\frac{c : \tau \in \Sigma \quad \{\alpha_1 : S_1 \ldots \alpha_n : S_n\} \in \mathtt{tvc}(\tau) \quad (\Sigma, \xi \vdash \tau_i : S_i)_i}{\Gamma \vdash c : \tau[\alpha_1 := \tau_1, \ldots, \alpha_1 := \tau_n]}\ \mathtt{CONST}$$

$$\frac{}{\Sigma, \Gamma \vdash x : \Gamma(x)}\ \mathtt{ASM} \qquad \frac{}{\Sigma, \Gamma \vdash ?x : \Gamma(?x)}\ \mathtt{ASM}$$

$$\frac{\Sigma, \Gamma \vdash e : \sigma \to \tau \quad \Sigma, \Gamma \vdash e' : \sigma}{\Sigma, \Gamma \vdash e\, e' : \tau}\ \mathtt{APP} \qquad \frac{\Sigma, \Gamma[x : \sigma] \vdash e : \tau}{\Sigma, \Gamma \vdash \lambda x^{\sigma}.\, e\ : \sigma \to \tau}\ \mathtt{ABS}$$

`tvc` computes an assignment of all type variables occurring in τ to the set of all constraints associated to it in τ.

Polymorphic Type Inferences (3)

The second judgement $\Sigma, \xi \vdash \tau : \{\kappa..\kappa\}$ infers if a type is admissible to a class κ:

Polymorphic Type Inferences (3)

The second judgement $\Sigma, \xi \vdash \tau : \{\kappa..\kappa\}$ infers if a type is admissible to a class κ:

$$\frac{(\Sigma, \xi \vdash \tau : \kappa_i)_{i \in \{1 \ldots n\}}}{\Sigma, \xi \vdash \tau : \{\kappa_1, \ldots, \kappa_n\}} \qquad \frac{\Sigma, \xi \vdash \tau : \{\kappa_1, \ldots, \kappa_n\} \quad i \in \{1 \ldots n\}}{\Sigma, \xi \vdash \tau : \kappa_i}$$

$$\frac{\xi(\alpha) = S}{\Sigma, \xi \vdash \alpha : S} \qquad \frac{(\kappa_1, \ldots, \kappa_n)\chi \mapsto \kappa \in \Delta \quad (\Sigma, \xi \vdash \tau_i : \kappa_i)_{i \in \{1 \ldots n\}}}{(\Sigma, \xi \vdash (\tau_1, \ldots, \tau_n)\chi : \kappa}$$

$$\frac{\Sigma, \xi \vdash \tau : \kappa_1 \quad \kappa_1 \leq \kappa_2}{\Sigma, \xi \vdash \tau : \kappa_2}$$

Polymorphic Type Inferences (3)

The second judgement $\Sigma, \xi \vdash \tau : \{\kappa..\kappa\}$ infers if a type is admissible to a class κ:

$$\frac{(\Sigma, \xi \vdash \tau : \kappa_i)_{i \in \{1 \ldots n\}}}{\Sigma, \xi \vdash \tau : \{\kappa_1, \ldots, \kappa_n\}} \qquad \frac{\Sigma, \xi \vdash \tau : \{\kappa_1, \ldots, \kappa_n\} \quad i \in \{1 \ldots n\}}{\Sigma, \xi \vdash \tau : \kappa_i}$$

$$\frac{\xi(\alpha) = S}{\Sigma, \xi \vdash \alpha : S} \qquad \frac{(\kappa_1, \ldots, \kappa_n)\chi \mapsto \kappa \in \Delta \quad (\Sigma, \xi \vdash \tau_i : \kappa_i)_{i \in \{1 \ldots n\}}}{(\Sigma, \xi \vdash (\tau_1, \ldots, \tau_n)\chi : \kappa}$$

$$\frac{\Sigma, \xi \vdash \tau : \kappa_1 \quad \kappa_1 \leq \kappa_2}{\Sigma, \xi \vdash \tau : \kappa_2}$$

Note that there are constraints for Δ which are ommitted here (see [NP95] for details).

The Logic Pure

Tactics = Programs building Meta-Theorems

When constructing proofs, there are

Tactics = Programs building Meta-Theorems

When constructing proofs, there are

- logic specific aspects (its rules)

Tactics = Programs building Meta-Theorems

When constructing proofs, there are

- logic specific aspects (its rules)
- logic independent aspects such as:

Tactics = Programs building Meta-Theorems

When constructing proofs, there are

- logic specific aspects (its rules)
- logic independent aspects such as:
  - binding and substitution
  - typing
  - managing side-conditions
  - managing assumptions and their discharge

Tactics = Programs building Meta-Theorems

When constructing proofs, there are

- logic specific aspects (its rules)
- logic independent aspects such as:
  - binding and substitution
  - typing
  - managing side-conditions
  - managing assumptions and their discharge

In textbooks, the focus is typically on the former and the latter were only described in informal "provisos".

Tactics = Programs building Meta-Theorems

When constructing proofs, there are

- logic specific aspects (its rules)
- logic independent aspects such as:
  - binding and substitution
  - typing
  - managing side-conditions
  - managing assumptions and their discharge

In textbooks, the focus is typically on the former and the latter were only described in informal "provisos".

- Using a metalogic Pure has two benefits:
  - shared implementations for the logic independent aspects, and
  - potential for "generic" proof procedures built on top of it.

Built on top of the syntactic language of the extended type class λ-calculus, Isabelle's meta-language Pure is implemented.

At least one type classes are assumed: $logic \in \kappa$. Moreover, at least two type constructors are assumed: $\texttt{prop}, _ \Rightarrow _ \in \chi$.

Logic Based on $\lambda^{\rightarrow}$

Then the signature Σ of Pure is defined as follows:

Logic Based on $\lambda^{\rightarrow}$

Then the signature Σ of Pure is defined as follows:

- $_ \Longrightarrow _ : \texttt{prop} \rightarrow \texttt{prop} \rightarrow \texttt{prop} \in \Sigma$,
- $_ \equiv _ : \alpha \rightarrow \alpha \rightarrow \texttt{prop} \in \Sigma$, and
- $\bigwedge _ : (\alpha \rightarrow \texttt{prop}) \rightarrow \texttt{prop} \in \Sigma$.

The _-notation is used to indicate infixes.

Terms of type $bool$ as in HOL, for example, were represented by a special constant $\texttt{Trueprop} :: bool \Rightarrow \texttt{prop}$. $\texttt{Trueprop}\,\phi$ corresponds to the pr-operator in the "Propositional Logic in LF" encoding or the textbook notation "$\vdash \phi$".

(Trueprop is usually supressed syntactically.)

The Format of thm

Isabelle's Pure is

- implemented in the style of the LCF system: meta-level rules are SML functions on **thm**, possibly raising exceptions,

The Format of thm

Isabelle's Pure is

- implemented in the style of the LCF system: meta-level rules are SML functions on **thm**, possibly raising exceptions,
- uses natural deduction:
  each **thm** may depend on meta-level assumptions:

$$\phi[\phi, \ldots, \phi]$$

The Format of thm

Isabelle's Pure is

- implemented in the style of the LCF system: meta-level rules are SML functions on **thm**, possibly raising exceptions,
- uses natural deduction:
  each **thm** may depend on meta-level assumptions:

$$\phi[\phi, \ldots, \phi]$$

- each **thm** has a signature $(\Sigma, \chi, \kappa, \Delta)$.

Asumption and Rules for $\Rightarrow$

Manipulating meta-level assumptions:

$$\frac{}{\phi[\phi]}\ \texttt{assume} \qquad \frac{\begin{array}{c}[\phi]\\ \vdots \\ \psi\end{array}}{\phi \Rightarrow \psi}\ \Rightarrow\text{-}I \qquad \frac{\phi \Rightarrow \psi \quad \phi}{\psi}\ \Rightarrow\text{-}E$$

Note that $\Rightarrow$-*I* is now understood fully operationally: ϕ is erased from the meta-level assumption list of the premise of $\Rightarrow$-*I*.

Rules for $\equiv$: Equivalence Relation

Rules:

$$\frac{\phi \Rightarrow \psi \quad \psi \Rightarrow \phi}{\phi \equiv \psi} \equiv -I \qquad \frac{\phi \equiv \psi \quad \phi}{\psi} \equiv -E$$

$$\frac{}{a \equiv a} \equiv\text{-}\mathit{refl} \qquad \frac{a \equiv b}{b \equiv a} \equiv\text{-}\mathit{symm}$$

$$\frac{a \equiv b \quad b \equiv c}{a \equiv c} \equiv\text{-}\mathit{trans}$$

Rules for $\equiv$: λ (i.e., α, β, η) Conversions

Compare to $=_{\alpha,\beta,\eta}$.

$$\frac{}{(\lambda x.a) \equiv (\lambda y.a[x \leftarrow y])}\ \alpha^* \qquad \frac{}{(\lambda x.a)b \equiv (a[x \leftarrow b])}\ \beta$$

$$\frac{f \equiv g}{f\ x \equiv g\ x}\ \eta^{**}$$

Side condition $*$: y is not free in a.

Side condition $**$: x is not free in f, g and the meta-level asumptions.

Conversion is built into the proof system, and Isabelle

routinely computes terms in α, β, η-normal-forms.

Note: These side conditions are directly implemented in the

SML code; in a way, this implemements similar side-conditions of object-logics once and for all.

Rules for $\equiv$: Abstraction, Combination

Rules

$$\frac{a \equiv b}{(\lambda x.a) \equiv (\lambda x.b)}\ \equiv\text{-}abstr^{*} \qquad \frac{f \equiv g \quad a \equiv b}{f\ a \equiv g\ b}\ \equiv\text{-}comb$$

Side condition $*$: x is not free in the meta-level assumptions.

Manipulating Meta-Variables

Rules:

$$\frac{\phi}{\phi[?x_1 := t_1, \ldots, ?x_n := t_n]}\ \texttt{instantiate}$$

`instantiate` can in fact also handle instantiations of type-meta variables, which we ignore throughout this presentation.

A somewhat exotic axiom scheme — traditionally treated as outside the core of Pure — introduces axiomatic type class invariants into the core logic:

$$\frac{}{\texttt{OFCLASS}(\alpha :: c, c_class)}\ \texttt{class_triv}$$

Rules for $\bigwedge$

Meta-quantification is formalized in higher-order abstract syntax: we write $\bigwedge x.\phi$ for $\bigwedge x.(\lambda x.\phi)$.

Rules for $\bigwedge$

Meta-quantification is formalized in higher-order abstract syntax: we write $\bigwedge x.\phi$ for $\bigwedge x.(\lambda x.\phi)$.

Rules:

$$\frac{\phi}{\bigwedge x.\phi}\ \bigwedge\text{-}I^* \qquad \frac{\bigwedge x.\phi}{\phi[x \leftarrow b]}\ \bigwedge\text{-}E$$

Rules for $\bigwedge$

Meta-quantification is formalized in higher-order abstract syntax: we write $\bigwedge x.\phi$ for $\bigwedge x.(\lambda x.\phi)$.

Rules:

$$\frac{\phi}{\bigwedge x.\phi}\ \bigwedge\text{-}I^* \qquad \frac{\bigwedge x.\phi}{\phi[x \leftarrow b]}\ \bigwedge\text{-}E$$

Side condition $*$: x is not free in meta-level assumptions. x may be a free variable or a meta-variable.

Rules for $\bigwedge$

Meta-quantification is formalized in higher-order abstract syntax: we write $\bigwedge x.\phi$ for $\bigwedge x.(\lambda x.\phi)$.

Rules:

$$\frac{\phi}{\bigwedge x.\phi}\ \bigwedge\text{-}I^* \qquad \frac{\bigwedge x.\phi}{\phi[x \leftarrow b]}\ \bigwedge\text{-}E$$

Side condition $*$: x is not free in meta-level assumptions. x may be a free variable or a meta-variable.

Note that combinations of $\bigwedge$-I^* and $\bigwedge$-E may therefore achieve the effect of replacing free variables by meta-variables.

What's different from HOL?

- no Falsum $\bot$,

What's different from HOL?

- no Falsum $\bot$,
- no classical :

What's different from HOL?

- no Falsum $\bot$,
- no classical :

  Pure is an intuitionistic fragment of HOL
- . . . what would be the consequences otherwise ?
- processes done by rtac / *rule* like:
  - lifting over assumptions
  - lifting over parameters

  are "rule schemes" implemented as tactical programs over Pure

Proof Objects

Although LCF-style systems were originally designed to avoid the construction of explicit proof-objects (as seen in LF), Isabelle has meanwhile a mechanism to "log" them during proof.

This has the following consequences:

- external proof-procedures can be used and recorded,
- proof-objects from extern provers may be imported,
- proof-objects of Isabelle can be checked externally.

How to generate Proof-Objects? (1)

theory ProofTest = Main:

ML{* *proofs* := *2* *}

lemma a1 : " a ⟶a" **by**(auto)

ML{* *ProofSyntax.print_proof_of false (thm "a1"); ** }

lemma a2 : " a ⟶b ⟶ a" **by**(auto)

ML{* *ProofSyntax.print_proof_of true (thm "a2"); ** }

How to generate Proof-Objects? (2)

```
equal_elim · _ · _ ·>
 (symmetric · _ · _ ·>
   (combination · Trueprop · _ · _·_·>( reflexive ·_)·>
     ( transitive · _ · _ · _ ·>
       (? ·> ( reflexive · _) ·>
         (ΛH: _.
           equal_elim · _ · _ ·>
            (symmetric · _ · _ ·>
              (combination · _ · _ · _ · _ ·>
                (combination · op ≡ ·_·_·_·>( reflexive ·_)·>
                  (Eq_TrueI · _ ·> H)) ·>
                ( reflexive · _))) ·>
            ( reflexive · _))) ·>
```

?))) ·>

Truel

How to generate Proof-Objects? (3)

```
equal_elim · _ · _ ·>
 (symmetric · _ · _ ·>
   (combination · Trueprop · _ · _ · _ ·> ( reflexive · _) ·>
     ( transitive · _ · _ · _ ·>
       (? ·> ( reflexive · _) ·>
         (Λ H: _.
           equal_elim · _ · _ ·>
            (symmetric · _ · _ ·>
              (combination · _ · _ · _ · _ ·>
                (combination · op≡·_·_·_·>( reflexive ·_)·>
                  ( transitive · _ · _ · _ ·>
                    (? ·> ( reflexive · _) ·>
                      (Λ Ha: _.
```

```
                    equal_elim · _ · _ ·>
                     (symmetric · _ · _ ·>
                       (combination · _ · _ · _ · _ ·>
                         (combination · op≡·_·_·_·>
                           ( reflexive · _) ·>
                           (Eq_TrueI · _ ·> H)) ·>
                         ( reflexive · _))) ·>
                     ( reflexive · _))) ·>
                 ?)) ·>
             ( reflexive · _))) ·>
         ( reflexive · _))) ·>
     ?))) ·>
TrueI
```

How to generate Proof-Objects

The proof-checker:

ProofChecker.thm_of_proof thy prf

returns a **thm** for a valid proof!

How to generate Proof-Objects

The proof-checker:

ProofChecker.thm_of_proof thy prf

returns a **thm** for a valid proof!

It consists of 100 lines of code (although reusing ca. 1000 lines of kernel code).

Conclusion on Isabelle's Metalogic

Conclusion on Isabelle's Metalogic

- The logic Pure and its proof system are small,

Conclusion on Isabelle's Metalogic

- The logic Pure and its proof system are small,
- Even resolution, and d-resolution are not built-in; they are tactics over Pure,

Conclusion on Isabelle's Metalogic

- The logic Pure and its proof system are small,
- Even resolution, and d-resolution are not built-in; they are tactics over Pure,
- Isabelle can log proofs in proof objects,

Conclusion on Isabelle's Metalogic

- The logic Pure and its proof system are small,
- Even resolution, and d-resolution are not built-in; they are tactics over Pure,
- Isabelle can log proofs in proof objects,
- If you don't trust Isabelle, check proof-objects !!!

More Detailed Explanations

The names of $\Rightarrow$, $\equiv$, and $\bigwedge$

- $\Longrightarrow$ is called meta-implication,
- $\equiv$ is called meta-equality, and
- $\bigwedge$ is called meta-universal-quantification.

Higer-Order Logic: Foundations

David Basin

Motivation

Motivation (1)

Higher-Order Logic (HOL) is:

Motivation (1)

Higher-Order Logic (HOL) is:

- an expressive foundation for

  mathematics: analysis, algebra, ...

  computer science: program correctness, hardware verification, ...

Motivation (1)

Higher-Order Logic (HOL) is:

- an expressive foundation for

  mathematics: analysis, algebra, ...

  computer science: program correctness, hardware verification, ...

- a logical framework for embedding languages/deductive systems.

  In contrast, there is no meta/object distinction then. Everything is defined within HOL. Reasoning is classical.

Motivation (2)

- Still modeling of problems is important (now in HOL)
  So is deriving relevant reasoning principles

Motivation (2)

- Still modeling of problems is important (now in HOL)
  So is deriving relevant reasoning principles
- We will use Isabelle/HOL

Motivation (2)

- Still modeling of problems is important (now in HOL)
  So is deriving relevant reasoning principles
- We will use Isabelle/HOL
  - Could forgo the use of a metalogic and employ alternatives, e.g., HOL system or PVS. Or use constructive alternatives such as Coq or Nuprl.

Motivation (2)

- Still modeling of problems is important (now in HOL)
  So is deriving relevant reasoning principles
- We will use Isabelle/HOL
  - Could forgo the use of a metalogic and employ alternatives, e.g., HOL system or PVS. Or use constructive alternatives such as Coq or Nuprl.
  - Choice depends on culture and application

Motivation (2)

- Still modeling of problems is important (now in HOL)
  So is deriving relevant reasoning principles
- We will use Isabelle/HOL
  - Could forgo the use of a metalogic and employ alternatives, e.g., HOL system or PVS. Or use constructive alternatives such as Coq or Nuprl.
  - Choice depends on culture and application

Motivation (3)

- HOL offers safety through strength

Motivation (3)

- HOL offers safety through strength

  Safety via conservative (definitional) extensions: functions, relations, inductive definitions, ...

Motivation (3)

- HOL offers safety through strength

  Safety via conservative (definitional) extensions: functions, relations, inductive definitions, ...
  - Extend theory with new constants and types defined by existing ones
  - Derive properties

Motivation (3)

- HOL offers safety through strength

  Safety via conservative (definitional) extensions: functions, relations, inductive definitions, ...

  - Extend theory with new constants and types defined by existing ones
  - Derive properties

- Contrast with

Motivation (3)

- HOL offers safety through strength

  Safety via conservative (definitional) extensions: functions, relations, inductive definitions, ...
  - Extend theory with new constants and types defined by existing ones
  - Derive properties
- Contrast with
  - Use of weak logics (e.g., propositional logic): can't define much
  - Axiomatic extensions: can lead to inconsistency

Motivation (3)

- HOL offers safety through strength

  Safety via conservative (definitional) extensions: functions, relations, inductive definitions, ...
  - Extend theory with new constants and types defined by existing ones
  - Derive properties
- Contrast with
  - Use of weak logics (e.g., propositional logic): can't define much
  - Axiomatic extensions: can lead to inconsistency

Bertrand Russel once likened the advantages of postulation over definition to the advantages of theft over honest toil!

Which Foundation?

- Set theory: the choice as basis for modern mathematics
  - ZFC (in Isabelle): impressive applications!
  - Bernays-Gödel: used for resolution since finitely axiomatizable

Which Foundation?

- Set theory: the choice as basis for modern mathematics
  - ZFC (in Isabelle): impressive applications!
  - Bernays-Gödel: used for resolution since finitely axiomatizable
- Set theories (both) distinguish between sets and classes
  - Consistency maintained as some collections are "too big" to be sets, e.g., class of all sets V is not a set.
  - A class cannot belong to another class (let alone a set)!

Which Foundation?

- Set theory: the choice as basis for modern mathematics
  - ZFC (in Isabelle): impressive applications!
  - Bernays-Gödel: used for resolution since finitely axiomatizable
- Set theories (both) distinguish between sets and classes
  - Consistency maintained as some collections are "too big" to be sets, e.g., class of all sets V is not a set.
  - A class cannot belong to another class (let alone a set)!
- HOL as alternative (Church 1940, Henkin 1950)
  - **Rationale:** one usually works with typed entities
  - Reasoning is then easier with support for types
  - Isabelle/HOL also supports like polymorphism and type classes!

HOL is weaker than ZF set theory, but for most applications this does not matter. If you prefer ML to Lisp, you will probably prefer HOL to ZF. (Paulson)

HOL — Why Higher-Order? (1)

1st-order: quantification over individuals (0th-order objects)

$$\forall x,\, y.\, R(x, y) \longrightarrow R(y, x)$$

2nd-order: quantification over predicates/functions

$$\begin{aligned} false &\equiv \forall P.\, P \\ P \wedge Q &\equiv \forall R.\, (P \longrightarrow Q \longrightarrow R) \longrightarrow R \end{aligned}$$

3rd-order: quantify over variables whose arguments are predicates

- Instead of defining:

$$subrel(R, S) \equiv \forall x.\, R(x) \longrightarrow S(x)$$

- Abstract and use:

$$\forall X.\, (X(R, S) \Leftrightarrow \forall x.\, R(x) \longrightarrow S(x)) \longrightarrow \ldots X(R', S') \ldots$$

HOL — Why Higher-Order? (2)

- Hierarchy: 4th, 5th, ... order logic

HOL — Why Higher-Order? (2)

- Hierarchy: 4th, 5th, ... order logic
- Omega-order logic: includes logics of all finite orders.

  Also called finite-type theory or higher-order logic

HOL — Why Higher-Order? (2)

- Hierarchy: 4th, 5th, ... order logic
- Omega-order logic: includes logics of all finite orders.
  Also called finite-type theory or higher-order logic
- Contrast: quantification over propositions versus functions
  - in LF we quantified over functions at all types

$$\forall f,\, g : N \to N. \forall x : n.\, f(x) =_N g(x) \Rightarrow f =_{N \to N} g$$

  - Result is a proposition and we cannot quantify over these!

HOL — Why Higher-Order? (2)

- Hierarchy: 4th, 5th, ... order logic
- Omega-order logic: includes logics of all finite orders. Also called finite-type theory or higher-order logic
- Contrast: quantification over propositions versus functions
  - in LF we quantified over functions at all types

$$\forall f,\, g : N \to N. \forall x : n.\, f(x) =_N g(x) \Rightarrow f =_{N \to N} g$$

  - Result is a proposition and we cannot quantify over these!

(LF sometimes called first-order! Better is minimal predicate logic with quantification over higher-types.)

Core-HOL: Syntax

Basic HOL Syntax (1)

- Types:

$$\tau \ ::= \ bool \mid ind \mid \tau \Rightarrow \tau$$

  - $bool$ and ind are also called o and i in literature [Chu40, And86]
  - Isabelle allows definitions of new type constructors (e.g., $A \times B$)
  - Isabelle supports polymorphic type definitions, e.g., $list(\alpha)$

Basic HOL Syntax (1)

- Types:

$$\tau \ ::= \ bool \mid ind \mid \tau \Rightarrow \tau$$

  - $bool$ and ind are also called o and i in literature [Chu40, And86]
  - Isabelle allows definitions of new type constructors (e.g., $A \times B$)
  - Isabelle supports polymorphic type definitions, e.g., $list(\alpha)$

- Terms: ($\mathcal{V}$ is set of variables, and $\mathcal{C}$ constants)

$$\mathcal{T} \ ::= \ \mathcal{V} \mid \mathcal{C} \mid (\mathcal{T}\mathcal{T}) \mid \lambda\mathcal{V}.\mathcal{T}$$

Terms are simply-typed. Terms of type $bool$ are called (well-formed) formulae.

Compare with Isabelle's Pure

Basic HOL Syntax (2)

- Constants are always supplied with types and include:

  $True, False : bool$

  $_ = _ : \tau \Rightarrow \tau \Rightarrow bool$ (for all types τ)

  $_ \rightarrow _ : bool \Rightarrow bool \Rightarrow bool$

  $\iota_ : (\tau \Rightarrow bool) \Rightarrow \tau$ (for all types τ)

Basic HOL Syntax (2)

- Constants are always supplied with types and include:

  $True, False : bool$

  $_ = _ : \tau \Rightarrow \tau \Rightarrow bool$ (for all types τ)

  $_ \rightarrow _ : bool \Rightarrow bool \Rightarrow bool$

  $\iota_ : (\tau \Rightarrow bool) \Rightarrow \tau$ (for all types τ)

Note that the description operator ιf yields the unique element x for which $f\,x$ is $True$, provided it exists. Otherwise, it yields an arbitrary value.

Basic HOL Syntax (2)

- Constants are always supplied with types and include:

  $True, False : bool$

  $_ = _ : \tau \Rightarrow \tau \Rightarrow bool$ (for all types τ)

  $_ \rightarrow _ : bool \Rightarrow bool \Rightarrow bool$

  $\iota_ : (\tau \Rightarrow bool) \Rightarrow \tau$ (for all types τ)

Note that the description operator ιf yields the unique element x for which $f\,x$ is $True$, provided it exists. Otherwise, it yields an arbitrary value.

Note that in Isabelle, the provisos "for all types τ" can be expressed by using polymorphic type variables α.

Core-HOL: Semantics

HOL Semantics

- Intuitively an extension of many-sorted semantics with functions
  - FOL: structure is domain and functions/relations

$$\langle \mathcal{D}, f_1, ..., f_k, r_1, ..., r_j \rangle$$

  - Many-sorted FOL: domains are sort-indexed

$$\langle \mathcal{D}_1, \ ... \mathcal{D}_n, f_1, ..., f_k, r_1, ..., r_j \rangle$$

    (if no relations then we have a heterogenous Algebra)
  - HOL extends idea: domain $\mathcal{D}$ is indexed by (infinitely many) types

- Our presentation ignores polymorphism on the object-logical level, it is treated on the meta-level, though (a version covering object-level parametric polymorphism is [GM93]).

Model Based on Universe of Sets $\mathcal{U}$

Definition 1 (Universe):

$\mathcal{U}$ is a collection of sets, fulfilling closure conditions:

Inhab: Each $X \in \mathcal{U}$ is nonempty set

Sub: If $X \in \mathcal{U}$ and $Y \neq \emptyset \subseteq X$, then $Y \in \mathcal{U}$

Prod: If $X, Y \in \mathcal{U}$ then $X \times Y \in \mathcal{U}$.

$X \times Y$ is Cartesian product, $\{\{x\}, \{x, y\}\}$ encodes (x, y)

Pow: If $X \in \mathcal{U}$ then $\mathcal{P}(X) = \{Y : Y \subseteq X\} \in \mathcal{U}$

Infty: $\mathcal{U}$ contains a distinguished infinite set I

Universe of Sets $\mathcal{U}$ (cont.)

- **Function space:**
  $X \Rightarrow Y$ is the set of (graphs of all total) functions from X to Y
  - For X and Y nonempty, $X \Rightarrow Y$ is nonempty and a subset of $\mathcal{P}(X \times Y)$
  - From closure conditions: $X, Y \in \mathcal{U}$ then so is $X \Rightarrow Y$.
- **Distinguished Sets:**
  from **Infty** and **Sub** there is (at least one) set
  Unit: A distinguished 1 element set $\{1\}$
  Bool: A distinguished 2 element set $\{T, F\}$.

Frames

Definition 2 (Frame):

A frame is a collection $\mathcal{D}_\alpha$ of sets, $\mathcal{D}_\alpha \in \mathcal{U}$, for $\alpha \in \tau$ where:

- $\mathcal{D}_{bool} = \{T, F\}$
- $\mathcal{D}_{Ind} = X$ where X is some infinite set of individuals
- $\mathcal{D}_{\alpha \Rightarrow \beta} \subseteq \mathcal{D}_\alpha \Rightarrow \mathcal{D}_\beta$, i.e., some collection of functions from D_α to D_β

Frames

Definition 2 (Frame):

A frame is a collection $\mathcal{D}_\alpha$ of sets, $\mathcal{D}_\alpha \in \mathcal{U}$, for $\alpha \in \tau$ where:

- $\mathcal{D}_{bool} = \{T, F\}$
- $\mathcal{D}_{Ind} = X$ where X is some infinite set of individuals
- $\mathcal{D}_{\alpha \Rightarrow \beta} \subseteq \mathcal{D}_\alpha \Rightarrow \mathcal{D}_\beta$, i.e., some collection of functions from D_α to D_β

Example: $\mathcal{D}_{bool \Rightarrow bool}$ is some nonempty subset of functions from $\{T, F\}$ to $\{T, F\}$. Some of these subsets contain, e.g., the identity function, others do not.

Interpretations

Definition 3 (Interpretation):

An interpretation $\langle \mathcal{D}_\alpha, \mathcal{J} \rangle$ is a frame $\mathcal{D}_\alpha$ and a denotation function $\mathcal{J}$ mapping each constant of type α to an element of $\mathcal{D}_\alpha$ where:

- $\mathcal{J}(True) = T$ and $\mathcal{J}(False) = F$
- $\mathcal{J}(=_{\alpha \Rightarrow \alpha \Rightarrow bool})$ is identity on $\mathcal{D}_\alpha$
- $\mathcal{J}(\rightarrow)$ denotes implication function over $\mathcal{D}_{bool}$.
  I.e., it sends $b, b' \in \{T, F\}$ to

$$b \longrightarrow b' = \begin{cases} F & \text{if } b = T \text{ and } b' = F \\ T & \text{otherwise} \end{cases}$$

- $\mathcal{J}(\iota_{(\alpha \Rightarrow bool) \Rightarrow \alpha}) \in (\mathcal{D}_\tau \Rightarrow \mathcal{D}_{bool}) \Rightarrow \mathcal{D}_\tau$ denotes the function

$$ch(f) = \begin{cases} a & \text{if } f = (\lambda x.x = a) \\ y & \text{otherwise} \end{cases}$$

for an arbitrary $y \in D_\alpha$ and an $f \in \mathcal{D}_\alpha \Rightarrow \mathcal{D}_{bool}$

- $\mathcal{J}(\iota_{(\alpha \Rightarrow bool) \Rightarrow \alpha}) \in (\mathcal{D}_\tau \Rightarrow \mathcal{D}_{bool}) \Rightarrow \mathcal{D}_\tau$ denotes the function

$$ch(f) = \begin{cases} a & \text{if } f = (\lambda x.x = a) \\ y & \text{otherwise} \end{cases}$$

  for an arbitrary $y \in D_\alpha$ and an $f \in \mathcal{D}_\alpha \Rightarrow \mathcal{D}_{bool}$

Note: the notion of an interpretation generalizes the notion of a structure to a higher-order setting.

Generalized Models

Definition 4 (Generalized Models):

An interpretation $\mathcal{M} = \langle \mathcal{D}_\alpha, \mathcal{J} \rangle$ is a (general) model for HOL iff there is a function $\mathcal{V}_A^{\mathfrak{M}}$ such that for all type-indexed families of substitutions $\sigma = \{\sigma_\alpha\}_{\alpha \in \tau}$ and terms, the following closure conditions hold:

1. $\mathcal{V}_A^{\mathfrak{M}}(x_\alpha) = \sigma(x_\alpha)$ (i.e., $\sigma_\alpha(x_\alpha)$)
2. $\mathcal{V}_A^{\mathfrak{M}}(c) = \mathcal{J}(c)$ for c a (primitive) constant
3. $\mathcal{V}_A^{\mathfrak{M}}(s_{\alpha \Rightarrow \beta} t_\alpha) = (\mathcal{V}_A^{\mathfrak{M}}(s))(\mathcal{V}_A^{\mathfrak{M}}(t))$
   i.e., the value of the function $\mathcal{V}_A^{\mathfrak{M}}(s)$ at the argument $\mathcal{V}_A^{\mathfrak{M}}(t)$

4. $\mathcal{V}_A^{\mathfrak{M}}(\lambda x_\alpha.\, t_\beta) =$ the function from $\mathcal{D}_\alpha$ into $\mathcal{D}_\beta$ whose value for each $z \in \mathcal{D}_\alpha$ is $\mathcal{V}_{\sigma[x \leftarrow z]}^{\mathcal{M}}(t)$

Generalized Models - Facts (1)

- **If** $\mathcal{M}$ is a general model and σ a substitution, **then** $\mathcal{V}_A^{\mathfrak{M}}$ is uniquely determined.
  $\mathcal{V}_A^{\mathfrak{M}}(t)$ is value of t in $\mathcal{M}$ wrt σ.
- Gives rise to the standard notion of satisfiability of formulae

$$\mathcal{V}_A^{\mathfrak{M}} \models \phi \text{iff } \mathcal{V}_A^{\mathfrak{M}}(\phi) = T$$

Generalized Models - Facts (2)

- Not all interpretations are general models.
- Closure conditions guarantee every well-formed formula has a value under every assignment, e.g.,

  closure under functions: identity function from $\mathcal{D}_\alpha$ to $\mathcal{D}_\alpha$ must always belong to $\mathcal{D}_{\alpha \Rightarrow \alpha}$ so that $\mathcal{V}_A^{\mathfrak{M}}(\lambda x_\alpha .\, x)$ defined.

  closure under application:
  - if $\mathcal{D}_N$ is natural numbers and
  - $\mathcal{D}_{N \Rightarrow N \Rightarrow N}$ contains addition function p where $p\, x\, y = x + y$
  - then $\mathcal{D}_{N \Rightarrow N}$ must contain $k\, x = 2x + 5$

    since $k = \mathcal{V}_A^{\mathfrak{M}}(\lambda x_N .\, f\,(f\, x\, x)\, y)$ where $\sigma(f) = p$ and $\sigma(y) = 5$.

Standard Models

Definition 5 (Standard Models):

A general model is a standard model iff for all $\alpha, \beta \in \tau$, $\mathcal{D}_{\alpha \Rightarrow \beta}$ is the set of all functions from $\mathcal{D}_\alpha$ to $\mathcal{D}_\beta$. A standard model is a general model, but not necessary vice versa.

Standard Models

Definition 5 (Standard Models):

A general model is a standard model iff for all $\alpha, \beta \in \tau$, $\mathcal{D}_{\alpha \Rightarrow \beta}$ is the set of all functions from $\mathcal{D}_\alpha$ to $\mathcal{D}_\beta$. A standard model is a general model, but not necessary vice versa.

We can now re-introduce HOL in Isabelle/Pure.

Isabelle/HOL

The syntax of the core-language is introduced by:

Trueprop	::	bool ⇒ prop	("(_)" 5)
Not	::	bool ⇒ bool	("¬ _" [40] 40)
True	::	bool	
False	::	bool	
If	::	[bool, 'a, 'a] ⇒ 'a	("(if _ then _ else _)")
The	::	('a ⇒ bool) ⇒ 'a	(**binder** "THE " 10)
All	::	('a ⇒ bool) ⇒ bool	(**binder** "∀ " 10)
Ex	::	('a ⇒ bool) ⇒ bool	(**binder** "∃ " 10)
=	::	['a, 'a] ⇒ bool	(**infixl** 50)
∧	::	[bool, bool] ⇒ bool	(infixr 35)
∨	::	[bool, bool] ⇒ bool	(infixr 30)
⟶	::	[bool, bool] ⇒ bool	(infixr 25)

The Axioms of HOL (1)

axioms

refl : "t = t"

subst: "⟦ s = t; P(s) ⟧ ⟹ P(t)"

ext: "(⋀x. f x = g x) ⟹ (λx. f x) = (λx. g x)"

impl: "(P ⟹ Q) ⟹ P ⟶ Q"

mp: "⟦ P ⟶ Q; P ⟧ ⟹ Q"

iff : "(P ⟶ Q) ⟶ (Q ⟶ P) ⟶ (P = Q)"

True_or_False : "(P = True) ∨ (P = False)"

the_eq_trivial :"(THE x. x = a) = (a::'a)

The Axioms of HOL (2)

Additionally, there is:

- universal α,β and η congruence on terms (implicitly),

The Axioms of HOL (2)

Additionally, there is:

- universal α,β and η congruence on terms (implicitly),
- the axiom of infinity,

The Axioms of HOL (2)

Additionally, there is:

- universal α,β and η congruence on terms (implicitly),
- the axiom of infinity,
- This is the entire basis!

Core-HOL: Meta-theoretic Properties

Meta-theoretic Properties of HOL

Theorem 1 (Soundness of HOL):

HOL is sound w.r.t. to generalized models.

$$\vdash_{HOL} \phi \Longrightarrow \mathcal{V}_A^{\mathfrak{M}} \models \phi$$

Meta-theoretic Properties of HOL

Theorem 1 (Soundness of HOL):

HOL is sound w.r.t. to generalized models.

$$\vdash_{HOL} \phi \Longrightarrow \mathcal{V}_A^{\mathfrak{M}} \models \phi$$

Theorem 2 (Completeness of HOL):

HOL is complete w.r.t. to generalized models.

$$\mathcal{V}_A^{\mathfrak{M}} \models \phi \Longrightarrow \vdash_{HOL} \phi$$

Meta-theoretic Properties of HOL

Theorem 1 (Soundness of HOL):

HOL is sound w.r.t. to generalized models.

$$\vdash_{HOL} \phi \Longrightarrow \mathcal{V}_A^{\mathfrak{M}} \models \phi$$

Theorem 2 (Completeness of HOL):

HOL is complete w.r.t. to generalized models.

$$\mathcal{V}_A^{\mathfrak{M}} \models \phi \Longrightarrow \vdash_{HOL} \phi$$

Theorem 3 (Completeness of HOL (without Infinity)):
HOL without the axiom of infinity is complete w.r.t. to standard models.

Theorem 3 (Completeness of HOL (without Infinity)):
HOL without the axiom of infinity is complete w.r.t. to standard models.

Theorem 4 (Incompleteness of HOL):
HOL is incomplete w.r.t. standard models.

Theorem 3 (Completeness of HOL (without Infinity)):
HOL without the axiom of infinity is complete w.r.t. to standard models.

Theorem 4 (Incompleteness of HOL):
HOL is incomplete w.r.t. standard models.

For the proofs, see [And86].

Core Definitions of HOL

True_def:	True	$\equiv ((\lambda x{::}bool.\ x) = (\lambda x.\ x))$
All_def:	All(P)	$\equiv (P = (\lambda x.\ True))$
Ex_def:	Ex(P)	$\equiv \forall Q.\ (\forall x.\ P\ x \longrightarrow Q) \longrightarrow Q$
False_def:	False	$\equiv (\forall P.\ P)$
not_def:	$\neg P$	$\equiv P \longrightarrow False$
and_def:	$P \wedge Q$	$\equiv \forall R.\ (P \longrightarrow Q \longrightarrow R) \longrightarrow R$
or_def:	$P \vee Q$	$\equiv \forall R.\ (P \longrightarrow R) \longrightarrow (Q \longrightarrow R) \longrightarrow R$
if_def:	If P x y	$\equiv THE\ z{::}'a.\ (P{=}True \longrightarrow z{=}x) \wedge (P{=}False \longrightarrow z{=}y)$

Definitions can be understood either

semantically: so-called shallow semantic embedding, or as

derived rules: in Isabelle, i.e., by their properties.

Conclusion

Conclusions

- HOL generalizes semantics of FOL
  - *bool* serves as type of propositions
  - Syntax/semantics allows for higher-order functions
- Logic is rather minimal: 8 rules, more-or-less obvious
- Logic is very powerful in terms of what we can represent/derive.
  - Other "logical" syntax
  - Rich theories via conservative extensions (topic for next few weeks!)

Bibliography

- M. J. C. Gordon and T. F. Melham, Introduction to HOL: A theorem proving environment for higher order logic, Cambridge University Press, 1993.
- Peter B. Andrews, An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof, Academic Press, 1986.
- Tobias Nipkow and Lawrence C. Paulson and Markus Wenzel, Isabelle/HOL — A Proof Assistant for Higher-Order Logic, Springer-Verlag, LNCS 2283, 2002.

Higer-Order Logic: Derived Rules

David Basin

Outline

Last lecture: Introduction to HOL

- Basic syntax and semantics
- Basic eight (or nine) axioms
- Definitions of $True$, $False$, $\wedge$, $\vee$, $\forall$ . . .

Today:

- Deriving rules for the defined constants
- Outlook on the rest of this course

Reminder: Different Syntaxes

Conceptual	vs.	Isabelle/PG notation
$\lambda x^{bool}.P(x)$		$\lambda \mathtt{x} :: \mathtt{bool}.\mathtt{P}$
$\forall x.\, P(x)$		"$All(\lambda \mathtt{x}.P\, x)$" = "$\forall x.P(x)$"
$\iota x.\, P(x)$		"$The(\lambda \mathtt{x}.P\, x)$" = "$\mathtt{THE}x.P(x)$"

We will be using all those forms as convenient.

Reminder: Definitions

True_def:	True	$\equiv ((\lambda x{::}\mathsf{bool}.\ x) = (\lambda x.\ x))$
All_def:	All(P)	$\equiv (P = (\lambda x.\ \mathsf{True}))$
Ex_def:	Ex(P)	$\equiv \forall Q.\ (\forall x.\ P\ x \longrightarrow Q) \longrightarrow Q$
False_def:	False	$\equiv (\forall P.\ P)$
not_def:	$\neg P$	$\equiv P \longrightarrow \mathsf{False}$
and_def:	$P \wedge Q$	$\equiv \forall R.\ (P \longrightarrow Q \longrightarrow R) \longrightarrow R$
or_def:	$P \vee Q$	$\equiv \forall R.\ (P \longrightarrow R) \longrightarrow (Q \longrightarrow R) \longrightarrow R$
if_def:	If P x y	$\equiv \mathsf{THE}\ z{::}'a.\ (P{=}\mathsf{True} \longrightarrow z{=}x) \wedge (P{=}\mathsf{False} \longrightarrow z{=}y)$

Derived Rules

The definitions can be understood as syntactic abbreviations.

Later, we will see that they are in fact conservative constant definitions.

We usually proceed as follows: first show a rule involving a constant, then replace the constant with its definition (if applicable), then show the derivation.

Equality

- Rule *sym*

$$\frac{}{}$$

Equality

- Rule *sym*

$$\frac{s = t}{t = s}\ \textit{sym}$$

Equality

- Rule *sym*

$$\frac{s = t}{t = s}\ \textit{sym}$$

- HOL rule s=t $\Longrightarrow$ t=s:

Equality

- Rule *sym* and ND derivation

$$\dfrac{s = t \quad \dfrac{}{s = s}\,\textit{refl}}{t = s}\,\textit{subst}$$

- HOL rule s=t $\Longrightarrow$ t=s: Proof:

```
lemma sym : "s=t ⟹t=s";
apply ( erule  subst );          (* P is λx.x=s *)
apply ( rule  refl  1);          (* s=s *)
done
```

Equality: Transitivity and Congruences

- Rule *trans*

$$\frac{s = t \quad r = s}{r = t}\ \textit{trans}$$

Equality: Transitivity and Congruences

- Rule *trans* and ND derivation

$$\frac{s = t \quad r = s}{r = t}\ \textit{subst}$$

HOL rule ⟦ r=s; s=t ⟧ ⟹ r=t

Equality: Transitivity and Congruences

- Rule *trans* and ND derivation

$$\frac{s = t \quad r = s}{r = t}\ \textit{subst}$$

HOL rule ⟦ r=s; s=t ⟧ ⟹ r=t

- Congruences (only HOL forms):
  - (f ::' a⇒'b) = g ⟹ f(x)=g(x) (funcong)

Equality: Transitivity and Congruences

- Rule *trans* and ND derivation

$$\frac{s = t \quad r = s}{r = t}\ \textit{subst}$$

HOL rule ⟦ r=s; s=t ⟧ ⟹ r=t

- Congruences (only HOL forms):
  - (f ::' a⇒'b) = g ⟹ f(x)=g(x) (funcong)
  - x=y ⟹f(x)=f(y) (argcong)

Equality: Transitivity and Congruences

- Rule *trans* and ND derivation

$$\frac{s = t \quad r = s}{r = t}\ \textit{subst}$$

HOL rule ⟦ r=s; s=t ⟧ ⟹ r=t

- Congruences (only HOL forms):
  - (f ::' a⇒'b) = g ⟹ f(x)=g(x) (funcong)
  - x=y ⟹f(x)=f(y) (argcong)

HOL proofs using *subst* and *refl*.

Equality of Booleans (*iffI*)

Rule *iffI*

Equality of Booleans (*iffI*)

Rule *iffI*

$$\frac{\begin{array}{c}[P]\\ \vdots \\ Q\end{array} \qquad \begin{array}{c}[Q]\\ \vdots \\ P\end{array}}{P = Q}\ \textit{iffI}$$

Equality of Booleans (*iffI*)

Rule *iffI*

$$\frac{\begin{array}{c}[P]\\ \vdots \\ Q\end{array} \qquad \begin{array}{c}[Q]\\ \vdots \\ P\end{array}}{P = Q}\ \textit{iffI}$$

HOL rule ⟦ P ⟹ Q; Q ⟹ P ⟧ ⟹ P=Q.

Equality of Booleans (*iffl*)

Rule *iffl* and ND derivation

$$\dfrac{\dfrac{\dfrac{}{(P \to Q) \to (Q \to P) \to (P = Q)}\,\textit{iff} \qquad \dfrac{\begin{matrix}[P]\\ \vdots \\ Q\end{matrix}}{P \to Q}\,\textit{impl}}{(Q \to P) \to P = Q}\,\textit{mp} \qquad \dfrac{\begin{matrix}[Q]\\ \vdots \\ P\end{matrix}}{Q \to P}\,\textit{impl}}{P = Q}\,\textit{mp}$$

HOL rule ⟦ P ⟹ Q; Q ⟹ P ⟧ ⟹ P=Q.

Equality of Booleans (*iffD2*)

Rule *iffD2*

$$P = Q$$

$$\frac{Q}{P}\ \textit{iffD2}$$

Equality of Booleans (*iffD2*)

Rule *iffD2* and ND derivation

$$\dfrac{\dfrac{P = Q}{Q = P}\,\textit{sym} \qquad Q}{P}\,\textit{subst}$$

HOL rule ⟦ P=Q; Q ⟧⟹P.

True

$$True = ((\lambda x^{bool}.x) = (\lambda x.x))$$

- Rule *Truel*

$$\frac{}{True}\ \textit{Truel}$$

True

$$True = ((\lambda x^{bool}.x) = (\lambda x.x))$$

- Rule *Truel*

$$\frac{}{(\lambda x.x) = (\lambda x.x)}\ \textit{Truel}$$

True

$$True = ((\lambda x^{bool}.x) = (\lambda x.x))$$

- Rule *Truel* and ND derivation

$$\frac{}{(\lambda x.x) = (\lambda x.x)}\ \textit{refl}$$

True

$$True = ((\lambda x^{bool}.x) = (\lambda x.x))$$

- Rule *TrueI* and ND derivation

$$\frac{}{(\lambda x.x) = (\lambda x.x)}\ \textit{refl}$$

- Rule *eqTrueE*

$$\frac{P = True}{P}\ \textit{eqTrueE}$$

True

$$True = ((\lambda x^{bool}.x) = (\lambda x.x))$$

- Rule *TrueI* and ND derivation

$$\frac{}{}\ \textit{refl}$$

- Rule *eqTrueE* and ND derivation

$$\frac{P = True \quad \frac{}{True}\ \textit{TrueI}}{P}\ \textit{iffD2}$$

HOL rule P=True $\Longrightarrow$P.

True (Cont.)

- Rule *eqTrueI*

$$\frac{P}{P = True}\ \textit{eqTrueI}$$

True (Cont.)

- Rule *eqTrueI* and ND derivation

$$\frac{\frac{}{True}\,{\scriptstyle TrueI} \qquad P}{P = True}\,{\scriptstyle iffI}$$

Note that 0 assumptions were discharged.

HOL rule P $\Longrightarrow$ P=True.

Universal Quantification

$\forall P = (P = (\lambda x.True))$

- Rule *alll*

$$\frac{}{\forall P}\ \textit{alll}$$

HOL rule $(\bigwedge$x. P(x)$) \Longrightarrow \forall$ x. P(x).

Universal Quantification

$\forall P = (P = (\lambda x.True))$

- Rule *alll*

$$\frac{}{P = \lambda x.True}\ \textit{alll}$$

HOL rule ($\bigwedge$x. P(x)) $\Longrightarrow \forall$ x. P(x).

Universal Quantification

$\forall P = (P = (\lambda x.True))$

- Rule *alll* and ND derivation

$$\dfrac{\dfrac{\bigwedge x.\, P(x)}{\bigwedge x.\, P(x) = True}\,\mathit{eqTruel}}{P = \lambda x.\, True}\,\mathit{ext}$$

HOL rule ($\bigwedge$x. P(x)) $\Longrightarrow \forall$ x. P(x).

Universal Quantification (Cont.)

- Rule *spec*

$$\frac{\forall P}{P(x)}\ \textit{spec}$$

Universal Quantification (Cont.)

- Rule *spec*

$$P = \lambda x.True$$

$$\frac{}{P(x)}\ \textit{spec}$$

Universal Quantification (Cont.)

- Rule *spec* and ND derivation

$$\dfrac{\dfrac{P = \lambda x.True}{P(x) = True}\,\textit{fun_cong}}{P(x)}\,\textit{eqTrueE}$$

HOL rule ∀ x ::' a. P(x) $\Longrightarrow$ P(x).

Note: Need universal quantification to reason about $False$ (since $False = (\forall P.P)$).

False

$$False = (\forall P.P)$$

- FalseI:

False

$False = (\forall P.P)$

- FalseI: No rule!
- Rule *FalseE*

$$\frac{False}{P}\ \textit{FalseE}$$

False

$False = (\forall P.P)$

- FalseI: No rule!
- Rule *FalseE*

$$\frac{\forall P.\, P}{P}\ \textit{FalseE}$$

False

$False = (\forall P.P)$

- FalseI: No rule!
- Rule *FalseE* and ND derivation

$$\frac{\forall P.\, P}{P}\ \textit{spec}$$

HOL rule False $\Longrightarrow$ P.

False (Cont.)

- Rule *False_neq_True*

$$False = True$$

$$\frac{}{P}\ \textit{False_neq_True}$$

False (Cont.)

- Rule *False_neq_True* and ND derivation

$$\dfrac{\dfrac{False = True}{False}\,\textit{eqTrueE}}{P}\,\textit{FalseE}$$

HOL rule False=True $\Longrightarrow$P.

-

False (Cont.)

- Rule *False_neq_True* and ND derivation

$$False = True$$

$$P$$

HOL rule False=True $\Longrightarrow$P.

- Similar:

$$\frac{True = False}{P}\ \textit{True_neq_False}$$

Negation

$\neg P = P \rightarrow False$

- Rule *notI*

$$\frac{\begin{array}{c}[P] \\ \vdots \\ False\end{array}}{\neg P}\ \textit{notI}$$

Negation

$\neg P = P \rightarrow False$

- Rule *notI*

$$\frac{\begin{array}{c}[P] \\ \vdots \\ False\end{array}}{P \rightarrow False}\ \textit{notI}$$

Negation

$\neg P = P \rightarrow False$

- Rule *notI* and ND derivation

$$\frac{\begin{array}{c}[P] \\ \vdots \\ False\end{array}}{P \rightarrow False}\ \textit{impI}$$

HOL rule (P $\Longrightarrow$ False) $\Longrightarrow \neg$ P.

Negation (Cont.)

- Rule *notE*

$$\frac{\neg P \qquad P}{R}\ \textit{notE}$$

Negation (Cont.)

- Rule *notE*

$$P \rightarrow False \quad P$$

$$\frac{}{R}\ \textit{notE}$$

Negation (Cont.)

- Rule *notE* and ND derivation

$$\dfrac{\dfrac{P \rightarrow False \quad P}{False}\,{\scriptstyle mp}}{R}\,{\scriptstyle FalseE}$$

HOL rule ⟦ ¬ P; P ⟧ ⟹ R.

Negation (Cont.)

- Rule *True_Not_False*

$$\frac{[\qquad\qquad\qquad]^1}{(True = False) \rightarrow False}\ \textit{True_Not_False}^{1}$$

Negation (Cont.)

- Rule *True_Not_False* and ND derivation

$$\dfrac{\dfrac{[True = False]^1}{False}\,\textit{True_neq_False}}{(True = False) \rightarrow False}\,\textit{notI}^1$$

HOL rule True $\neq$ False.

Existential Quantification

- $\mathrm{Ex}(P) \equiv \forall Q.\ (\forall x.\ P\ x \longrightarrow Q) \longrightarrow Q$

Existential Quantification

- $\mathrm{Ex}(P) \equiv \forall Q.\ (\forall x.\ P\ x \longrightarrow Q) \longrightarrow Q$
- $P(x) \Longrightarrow \exists\ x ::' a . P(x) \quad (exI)$

Existential Quantification

- Ex(P) $\equiv \forall$ Q. ($\forall$ x. P x $\longrightarrow$ Q) $\longrightarrow$ Q
- P(x) $\Longrightarrow \exists$ x ::' a . P(x) (*exI*)

$$\dfrac{\dfrac{\dfrac{Px \qquad \dfrac{[\forall y.P(y) \to Q]}{P(x) \to Q}\,\textit{spec}}{Q}\,\textit{mp}}{(\forall y.P(y) \to Q) \to Q)}\,\textit{impI}}{\forall Q.(\forall x.P(x) \to Q) \to Q)}\,\textit{allI}$$

- $[\![\ \exists\, x ::' a . P(x);\ \bigwedge x.\ P(x) \Longrightarrow Q\ ]\!] \Longrightarrow Q \quad (exE)$

- $[\![\ \exists\ x ::' a . P(x);\ \bigwedge x.\ P(x) \Longrightarrow Q\ ]\!] \Longrightarrow Q \quad (exE)$

$$\dfrac{\dfrac{\forall Q.((\forall y.P(y) \to Q) \to Q)}{(\forall y.P(y) \to Q) \to Q}\ \textit{spec} \qquad \dfrac{\bigwedge x.\ \dfrac{[P(x)]}{Q}\ \textit{impI}}{\forall y.P(y) \to Q}\ \textit{allI}}{Q}\ \textit{mp}$$

Conjunction

$$P \wedge Q = \forall R.(P \to Q \to R) \to R$$

- Rule *conjI*

$$\frac{P \qquad Q}{P \wedge Q}\ \textit{conjI}$$

Conjunction

$$P \wedge Q = \forall R.(P \to Q \to R) \to R$$

- Rule *conjI*

$$P$$

$$Q$$

$$\frac{}{\forall R.(P \to Q \to R) \to R}\ \textit{conjI}$$

Conjunction

$$P \wedge Q = \forall R.(P \to Q \to R) \to R$$

- Rule *conjl* and ND derivation

$$\dfrac{\dfrac{\dfrac{\dfrac{[P \to Q \to R]^1 \quad P}{Q \to R}\,mp \qquad Q}{R}\,mp}{(P \to Q \to R) \to R}\,impl^1}{\forall R.(P \to Q \to R) \to R}\,alll$$

HOL rule $[\![$ P; Q $]\!] \Longrightarrow$ P $\wedge$ Q.

Conjunction (Cont.)

- Rule *conjEL*

$$\frac{P \wedge Q}{P}\ \textit{conjEL}$$

Conjunction (Cont.)

- Rule *conjEL*

$$\forall R.(P \to Q \to R) \to R$$

$$\frac{}{P}\ \textit{conjEL}$$

Conjunction (Cont.)

- Rule *conjEL* and ND derivation

$$\dfrac{\dfrac{\forall R.(P \to Q \to R) \to R}{(P \to Q \to P) \to P}\,\textit{spec} \qquad \dfrac{\dfrac{[P]^1}{Q \to P}\,\textit{impl}}{P \to Q \to P}\,\textit{impl}^1}{P}\,\textit{mp}$$

HOL rule P $\wedge$ Q $\Longrightarrow$P.

Conjunction (Cont.)

- $P \wedge Q \Longrightarrow Q$" (*conjER*)

Conjunction (Cont.)

- $P \wedge Q \Longrightarrow Q$" (*conjER*)
- $[\![P \wedge Q; [\![P; Q]\!] \Longrightarrow R]\!] \Longrightarrow R$ (*conjE*) (rule analogous to *disjE*)

Disjunction

$$P \vee Q = \forall R.(P \to R) \to (Q \to R) \to R$$

- Rule *disjIL*

$$\frac{P}{P \vee Q}\ \textit{disjIL}$$

Disjunction

$$P \vee Q = \forall R.(P \to R) \to (Q \to R) \to R$$

- Rule *disjIL*

$$P$$

$$\frac{}{\forall R.(P \to R) \to (Q \to R) \to R}\ \textit{disjIL}$$

Disjunction

$$P \vee Q = \forall R.(P \to R) \to (Q \to R) \to R$$

- Rule *disjIL* and ND derivation

$$\dfrac{\dfrac{\dfrac{\dfrac{[P \to R]^1 \quad P}{R}\,\mathit{mp}}{(Q \to R) \to R}\,\mathit{impl}}{(P \to R) \to (Q \to R) \to R}\,\mathit{impl}^1}{\forall R.(P \to R) \to (Q \to R) \to R}\,\mathit{alll}$$

HOL rule P $\Longrightarrow$ P $\vee$ Q.

Disjunction (Cont.)

- $Q \Longrightarrow P \vee Q$ (*disjIR*) similar
- Rule *disjE*

$$\frac{P \vee Q \qquad \begin{array}{c} P \\ \vdots \\ R \end{array} \qquad \begin{array}{c} Q \\ \vdots \\ R \end{array}}{R}\ \textit{disjE}$$

Disjunction (Cont.)

- $Q \Longrightarrow P \vee Q$ (*disjIR*) similar
- Rule *disjE*

$$\forall R.(P \to R) \to (Q \to R) \to R \qquad \frac{\begin{array}{c} P \\ \vdots \\ R \end{array} \qquad \begin{array}{c} Q \\ \vdots \\ R \end{array}}{R}\ \textit{disjE}$$

Disjunction (Cont.)

- $Q \Longrightarrow P \vee Q$ (*disjIR*) similar
- Rule *disjE* and ND derivation

$$\dfrac{\dfrac{\dfrac{\forall R.(P \to R) \to (Q \to R) \to R}{(P \to R) \to (Q \to R) \to R}\,\textit{spec} \qquad \dfrac{\begin{array}{c}P\\ \vdots\\ R\end{array}}{P \to R}\,\textit{impl}}{(Q \to R) \to R}\,\textit{mp} \qquad \dfrac{\begin{array}{c}Q\\ \vdots\\ R\end{array}}{Q \to R}\,\textit{impl}}{R}\,\textit{mp}$$

HOL rule $[\![\ P \vee Q;\ P \Longrightarrow R;\ Q \Longrightarrow R\ ]\!] \Longrightarrow R.$

Disjunction (Cont.)

- $Q \Longrightarrow P \vee Q$ (*disjIR*) similar
- Rule *disjE* and ND derivation

$$\dfrac{\dfrac{\dfrac{\forall R.(P \to R) \to (Q \to R) \to R}{(P \to R) \to (Q \to R) \to R}\,\textit{spec} \qquad \dfrac{\begin{array}{c}P\\ \vdots\\ R\end{array}}{P \to R}\,\textit{impl}}{(Q \to R) \to R}\,\textit{mp} \qquad \dfrac{\begin{array}{c}Q\\ \vdots\\ R\end{array}}{Q \to R}\,\textit{impl}}{R}\,\textit{mp}$$

HOL rule ⟦ P ∨ Q; P ⟹ R; Q ⟹ R ⟧ ⟹ R.

- $P \vee \neg P$ (*excluded middle*). Follows using *tof*.

Miscellaneous Definitions

Typical example (if-then-else):

If P x y $\equiv$ THE z. (P=True $\longrightarrow$ z=x) $\wedge$
(P=False $\longrightarrow$ z=y)

The way rules are derived should now be clear. E.g.,

$$\frac{P = True}{If\ P\ x\ y = x} \qquad \frac{P = False}{If\ P\ x\ y = y}$$

Summary on Deriving Rules

HOL is very powerful in terms of what we can represent/derive:

- All well-known inference rules can be derived.
- Other "logical" syntax (e.g. if-then-else) can be defined.
- Rich theories can be obtained by a method we see next lecture.

Mathematics and Software Engineering in HOL

In the weeks to come, we will see how Isabelle/HOL can be used as foundation for mathematics and software engineering.

Outline:

- The central method for making HOL scale up: conservative extensions (< 1 week)
- How the different parts of mathematics are encoded in the Isabelle/HOL library (several weeks)
- How software systems are embedded in Isabelle/HOL

(several weeks)

Outlook on Mathematics

After some historical background, we will look at how central parts of mathematics are encoded as Isabelle/HOL theories:

- Orders and sets
- Fixpoints, induction, and recursion
- Arithmetic
- Datatypes

Outlook on Software Engineering

Some weeks from now, we will look at case studies of how HOL can be applied in software engineering, i.e. how software systems can be embedded in Isabelle/HOL:

- Foundations, functional languages and denotational semantics
- Imperative languages, Hoare logic
- Z and data-refinement, CSP and process-refinement
- Object-oriented languages (Java-Light . . .)

Of the last three items, we want to treat only one in depth, depending on the audience's preferences.

Conservative Extensions: Motivation

But first, conservative extensions.

Stage of our course before studying HOL:

- fairly small theories,
- "intuitive" models, (e.g. naïve set theory),
- but inconsistent (due to foundational problems).

How can we ever hope to apply these techniques to software engineering?

What Is Needed for Scaling up?

Let's try to apply well-known structuring techniques:

What Is Needed for Scaling up?

Let's try to apply well-known structuring techniques:

Modularization

What Is Needed for Scaling up?

Let's try to apply well-known structuring techniques:

Modularization

Reuse

What Is Needed for Scaling up?

Let's try to apply well-known structuring techniques:

Modularization

Reuse
Safe, well-understood
integration mechanisms

What Is Needed for Scaling up?

Let's try to apply well-known structuring techniques:

Known mechanisms:

Modularization	$\Rightarrow$	(Parameterized) theories, (class) polymorphism
Reuse	$\Rightarrow$	Libraries, retrieval utilities
Safe, well-understood integration mechanisms	$\Rightarrow$	Persistent parametric theories, conservative theory extensions

What Is Needed for Scaling up?

Let's try to apply well-known structuring techniques:

Known mechanisms, of which Isabelle implements:

Modularization	$\Rightarrow$	~~(Parameterized)~~ theories, (class) polymorphism
Reuse	$\Rightarrow$	Libraries, retrieval utilities
Safe, well-understood integration mechanisms	$\Rightarrow$	~~Persistent parametric theories,~~ conservative theory extensions

What Is Needed for Scaling up?

Conservative theory extensions

Topic of next lecture.

More Detailed Explanations

RC

RC stands for refinement calculus.

Z, CSP

Z and CSP are specification languages. CSP stands for communicating sequential processes.

Persistence

Persistent theories play a role in the prover PVS.

Higher-Order Logic: Conservative Extensions

Burkhart Wolff

Outline

In the previous lecture, we have derived all well-known inference rules. There is now the need to scale up. Today we look at conservative theory extensions, an important method for this purpose.

Outline

In the previous lecture, we have derived all well-known inference rules. There is now the need to scale up. Today we look at conservative theory extensions, an important method for this purpose.

In the weeks to come, we will look at how mathematics is encoded in the Isabelle/HOL library.

Conservative Theory Extensions: Basics

Basic definitions (c.f. [GM93]):

Definition 6 (theory):

A (syntactic) theory T is a triple (χ, Σ, A), where χ is a type signature, Σ a signature and A and a set of axioms.

Definition 7 (theory extension):

A theory $T' = (\chi', \Sigma', A')$ is an extension of a theory $T = (\chi, \Sigma, A)$ iff $\chi \subseteq \chi'$ and $\Sigma \subseteq \Sigma'$ and $A \subseteq A'$.

Definitions (Cont.)

Definition 8 (conservative extension):

A theory extension $T' = (\chi', \Sigma', A')$ of a theory $T = (\chi, \Sigma, A)$ is conservative iff for the set of provable formulas Th we have

$$Th(T) = Th(T') \mid_{\Sigma},$$

where $\mid_{\Sigma}$ filters away all formulas not belonging to Σ.

Counterexample:

$$\frac{}{\forall f :: \alpha \Rightarrow \alpha.\ Y\ f = f\ (Y\ f)}\ \text{fix}$$

Consistency Preserved

Corollary 1 (consistency):

If T' is a conservative extension of T, then

$$False \notin Th(T) \Rightarrow False \notin Th(T').$$

Syntactic Schemata for Conservative Extensions

- Constant definition
- Type definition
- Constant specification
- Type specification

Will look at first two schemata now.

For the other two see [GM93].

Constant Definition

Definition 9 (constant definition):

A theory extension $T' = (\chi', \Sigma', A')$ of a theory $T = (\chi, \Sigma, A)$ is a constant definition, iff

- $\chi' = \chi$ and $\Sigma' = \Sigma \cup \{c :: \tau\}$, where $c \notin dom(\Sigma)$;
- $A' = A \cup \{c = E\}$;
- E does not contain c and is closed;
- no subterm of E has a type containing a type variable that is not contained in the type of c.

Constant Definitions Are Conservative

Lemma 1 (constant definitions):

Constant definitions are conservative.

Proof Sketch:

- $Th(T) \subseteq Th(T') \mid_\Sigma$: trivial.
- $Th(T) \supseteq Th(T') \mid_\Sigma$: let π' be a proof for $\phi \in Th(T') \mid_\Sigma$. We unfold any subterm in π' that contains c via $c = E$ into π. Then π must be a proof in T, implying $\phi \in Th(T)$.

Side Conditions

Where are those side conditions needed? What goes wrong?

Side Conditions

Where are those side conditions needed? What goes wrong? Very simple example: Let $E \equiv \exists x :: \alpha y :: \alpha.\ x \neq y$ and suppose σ is a type inhabited by only one term, and τ is a type inhabited by at least two terms. Then we would have:

$$\begin{array}{ll} & c = c \qquad \text{holds by } \textit{refl} \\ \Longrightarrow & (\exists x :: \sigma\, y :: \sigma.\, x \neq y) = (\exists x :: \tau\, y :: \tau.\, x \neq y) \\ \Longrightarrow & False = True \\ \Longrightarrow & False \end{array}$$

This explains definition of $True$. Other (standard) example later.

Constant Definition: Examples

Definitions of $True$, $False$, $\wedge$, $\vee$, $\forall$ revisited.

True_def:	True	$\equiv ((\lambda x{::}bool.\ x) = (\lambda x.\ x))$
All_def:	All(P)	$\equiv (P = (\lambda x.\ True))$
Ex_def:	Ex(P)	$\equiv \forall Q.\ (\forall x.\ P\ x \longrightarrow Q) \longrightarrow Q$
False_def:	False	$\equiv (\forall P.\ P)$
not_def:	$\neg P$	$\equiv P \longrightarrow False$
and_def:	$P \wedge Q$	$\equiv \forall R.\ (P \longrightarrow Q \longrightarrow R) \longrightarrow R$
or_def:	$P \vee Q$	$\equiv \forall R.\ (P \longrightarrow R) \longrightarrow (Q \longrightarrow R) \longrightarrow R$
if_def:	If P x y	$\equiv THE\ z{::}'a.\ (P{=}True \longrightarrow z{=}x) \wedge (P{=}False \longrightarrow z{=}y)$

Recall that All(P) is syntactically equivalent to $\forall$ x. P x, Ex(P) equivalent to $\exists$ x. P x.

More Constant Definitions in Isabelle

let−**in**−, if−then−else, unique existence:

consts

```
Let  :: ['a, 'a ⇒ 'b] ⇒ 'b
If   :: [bool, 'a, 'a] ⇒ 'a
```

defs

```
Let_def "Let s f ≡ f(s)"
if_def  "If P x y ≡ THE z::'a.(P=True⇒z=x) ∧
                              (P=False⇒z=y)"
Ex1_def "Ex1(P) ≡ ∃x. P(x) ∧ (∀y. P(y) ⇒ y=x)"
```

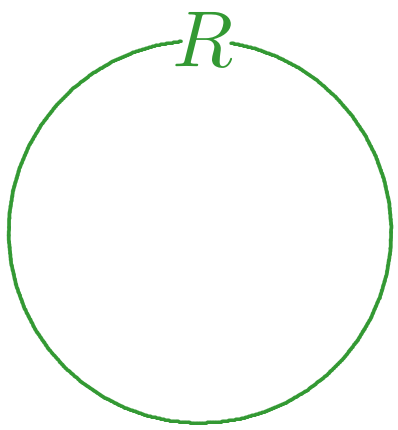
Recall: ⇒ is function type arrow; also recall syntax for

[...] ⇒ ... .

Type Definitions

Type definitions, explained intuitively: we have

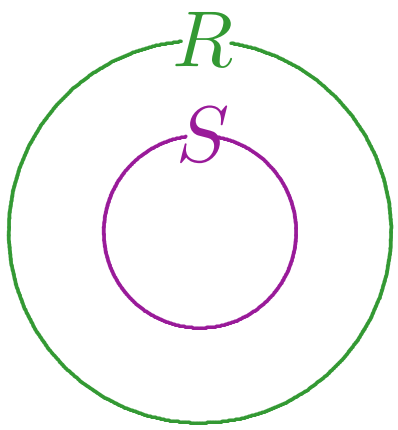
- an existing type R;



Type Definitions

Type definitions, explained intuitively: we have

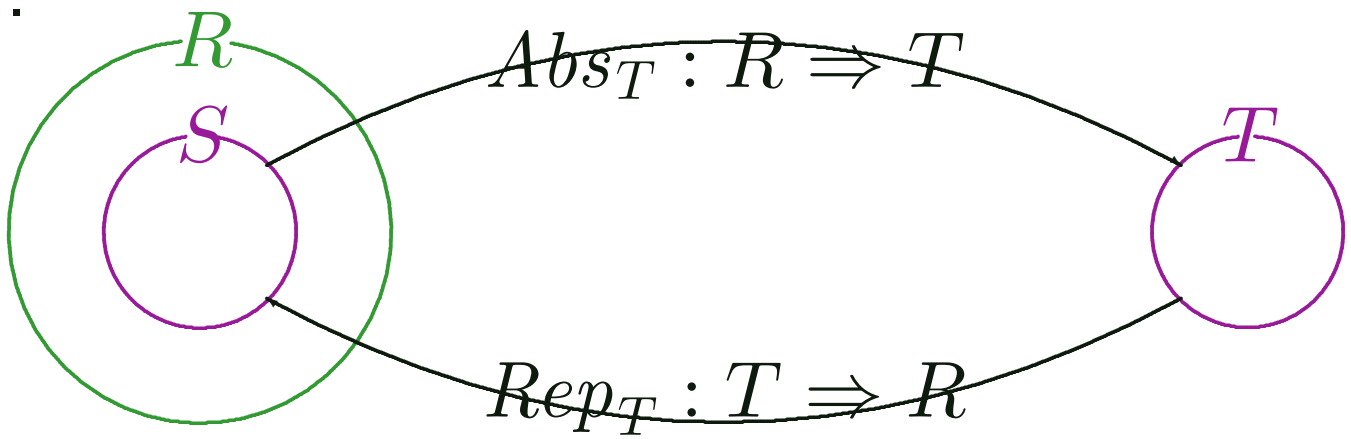
- an existing type R;
- a predicate $S : R \Rightarrow bool$, defining a non-empty "subset" of R;



Type Definitions

Type definitions, explained intuitively: we have

- an existing type R;
- a predicate $S : R \Rightarrow bool$, defining a non-empty "subset" of R;
- axioms stating an isomorphism between S and the new type T.



Type Definition: Definition

Definition 10 (type definition):

Assume a theory $Th = (\chi, \Sigma, A)$ and a type R and a term S such that $\Sigma \vdash S : R \Rightarrow bool$.

A theory extension $Th' = (\chi', \Sigma', A')$ of Th is a type definition for type T (where T fresh), iff

$$\begin{array}{rcll} \chi' & = & \chi & \uplus \ \{T\}, \\ \Sigma' & = & \Sigma & \cup \ \{Abs_T : R \Rightarrow T, Rep_T : T \Rightarrow R\} \\ A' & = & A & \cup \ \{\forall x. Abs_T(Rep_T\, x) = x, \\ & & & \quad \forall x. S\, x \Rightarrow Rep_T(Abs_T\, x) = x\} \end{array}$$

Proof obligation $\exists x.\, S\, x$ can be proven inside HOL!

Type Definitions Are Conservative

Lemma 2 (type definitions):

Type definitions are conservative.

Proof see [GM93, pp.230].

HOL Is Rich Enough!

This may seem fishy: if a new type is always isomorphic to a subset of an existing type, how is this construction going to lead to a "rich" collection of types for large-scale applications?

HOL Is Rich Enough!

This may seem fishy: if a new type is always isomorphic to a subset of an existing type, how is this construction going to lead to a "rich" collection of types for large-scale applications?

But in fact, due to ind and $\Rightarrow$, the types in HOL are already very rich.

HOL Is Rich Enough!

This may seem fishy: if a new type is always isomorphic to a subset of an existing type, how is this construction going to lead to a "rich" collection of types for large-scale applications?

But in fact, due to ind and $\Rightarrow$, the types in HOL are already very rich.

We now give three examples revealing the power of type definitions.

Example: Typed Sets

General scheme,

$$
\begin{array}{rcl}
\chi' & = & \chi \uplus \{T\ \}, \\
\Sigma' & = & \Sigma \cup \{Abs_T\ : R \Rightarrow T\ , \\
 & & \quad Rep_T\ : T \Rightarrow R\ \} \\
A' & = & A \cup \{\forall x. Abs_T\ (Rep_T\ x) = x, \\
 & & \quad \forall x. S\, x \Rightarrow Rep_T\ (Abs_T\ x) = x\}
\end{array}
$$

Example: Typed Sets

General scheme, substituting $R \equiv \alpha \Rightarrow bool$ (α is any type variable),

$$
\begin{array}{rcl}
\chi' & = & \chi \uplus \{T\ \}, \\
\Sigma' & = & \Sigma \cup \{Abs_T\ : (\alpha \Rightarrow bool) \Rightarrow T\ \ , \\
 & & \quad Rep_T\ : T\ \ \Rightarrow (\alpha \Rightarrow bool)\} \\
A' & = & A \cup \{\forall x. Abs_T\ (Rep_T\ x) = x, \\
 & & \quad \forall x. S\, x\ \ \Rightarrow Rep_T\ (Abs_T\ x) = x\}
\end{array}
$$

Example: Typed Sets

General scheme, substituting $R \equiv \alpha \Rightarrow bool$ (α is any type variable), $T \equiv \alpha\ set$ (or set),

$$
\begin{array}{rcll}
\chi' & = & \chi & \uplus\ \{set\}, \\
\Sigma' & = & \Sigma & \cup\ \{Abs_{set} : (\alpha \Rightarrow bool) \Rightarrow \alpha\ set, \\
 & & & \quad Rep_{set} : \alpha\ set \Rightarrow (\alpha \Rightarrow bool)\} \\
A' & = & A & \cup\ \{\forall x. Abs_{set}(Rep_{set}\, x) = x, \\
 & & & \quad \forall x. S\, x \quad \Rightarrow Rep_{set}(Abs_{set}\, x) = x\}
\end{array}
$$

Example: Typed Sets

General scheme, substituting $R \equiv \alpha \Rightarrow bool$ (α is any type variable), $T \equiv \alpha\ set$ (or set), $S \equiv \lambda x :: \alpha \Rightarrow bool.\, True$

$$
\begin{array}{rcll}
\chi' & = & \chi & \uplus\ \{set\}, \\
\Sigma' & = & \Sigma & \cup\ \{Abs_{set} : (\alpha \Rightarrow bool) \Rightarrow \alpha\ set, \\
 & & & \quad Rep_{set} : \alpha\ set \Rightarrow (\alpha \Rightarrow bool)\} \\
A' & = & A & \cup\ \{\forall x. Abs_{set}(Rep_{set}\, x) = x, \\
 & & & \quad \forall x. True \Rightarrow Rep_{set}(Abs_{set}\, x) = x\}
\end{array}
$$

Example: Typed Sets

General scheme, substituting $R \equiv \alpha \Rightarrow bool$ (α is any type variable), $T \equiv \alpha\ set$ (or set), $S \equiv \lambda x :: \alpha \Rightarrow bool.\, True$

$$
\begin{array}{rcll}
\chi' & = & \chi & \uplus\ \{set\}, \\
\Sigma' & = & \Sigma & \cup\ \{Abs_{set} : (\alpha \Rightarrow bool) \Rightarrow \alpha\ set, \\
 & & & \quad Rep_{set} : \alpha\ set \Rightarrow (\alpha \Rightarrow bool)\} \\
A' & = & A & \cup\ \{\forall x. Abs_{set}(Rep_{set}\, x) = x, \\
 & & & \quad \forall x. \qquad Rep_{set}(Abs_{set}\, x) = x\}
\end{array}
$$

Simplification since $S \equiv \lambda x.\, True$. Proof obligation: $(\exists x.\, S\, x)$ trivial since $(\exists x.\, True) = True$. Inhabitation is crucial!

Sets: Remarks

Any function $r : \tau \Rightarrow bool$ can be interpreted as a set of τ; r is called characteristic function. That's what $Abs_{set}\ r$ does; Abs_{set} is a wrapper saying "interpret r as set".

Sets: Remarks

Any function $r : \tau \Rightarrow bool$ can be interpreted as a set of τ; r is called characteristic function. That's what $Abs_{set}\ r$ does; Abs_{set} is a wrapper saying "interpret r as set".

$S \equiv \lambda x.True$ and so S is trivial in this case.

More Constants for Sets

For convenient use of sets, we define more constants:

$$\begin{array}{rcl} \{x \mid f\,x\} & \cong & Collect\ f = Abs_{set}\,f \\ x \in A & = & (Rep_{set}\ A)\ x \\ A \cup B & = & \{x \mid x \in A \vee x \in B\} \\ & \vdots & \end{array}$$

Consistent set theory adequate for most of mathematics and computer science !

Here, sets are just an example to demonstrate type definitions. Later we study them for their own sake.

Example: Pairs

Consider type $\alpha \Rightarrow \beta \Rightarrow bool$. We can regard a term $f : \alpha \Rightarrow \beta \Rightarrow bool$ as a representation of the pair (a, b), where $a :: \alpha$ and $b :: \beta$, iff $f\,x\,y$ is true exactly for $x = a$ and $y = b$. Observe:

- For given a and b, there is exactly one such f (namely, $\lambda x :: \alpha\, y :: \beta . x = a \wedge y = b$).
- Some functions of type $\alpha \Rightarrow \beta \Rightarrow bool$ represent pairs and others don't (e.g., the function $\lambda xy. True$ does not represent a pair). The ones that do are exactly the ones that have the form $\lambda x :: \alpha\, y :: \beta .\, x = a \wedge y = b$, for some a and b.

Type Definition for Pairs

This gives rise to a type definition where S is non-trivial:

$$
\begin{aligned}
R &\equiv \alpha \Rightarrow \beta \Rightarrow bool \\
S &\equiv \lambda f :: \alpha \Rightarrow \beta \Rightarrow bool. \\
&\qquad \exists ab.\, f = \lambda x :: \alpha\, y :: \beta . x = a \wedge y = b \\
T &\equiv \alpha \times \beta \qquad\qquad (\times \text{ infix})
\end{aligned}
$$

It is convenient to define a constant Pair_Rep (not to be confused with $Rep_\times$) as follows: Then
Pair_Rep a b = λx ::' a y ::' b. x=a $\wedge$y=b.

Implementation in Isabelle

Isabelle provides a special syntax for type definitions:

typedef (T)
(typevars) T' = "{x. A(x)}"

How is this linked to our scheme:

- the new type is called T';
- R is the type of x (inferred);
- S is $\lambda x.\, A\, x$;
- constants `Abs_`T and `Rep_`T are automatically generated.

Isabelle Syntax for Pair Example

constdefs

Pair_Rep :: ['a, 'b] ⇒ ['a, 'b] ⇒ bool

"Pair_Rep ≡ (λ a b. λ x y. x=a ∧y=b)"

typedef (Prod)

('a, 'b) "∗" (infixr 20) =

"{f.∃ a b. f=Pair_Rep(a::'a)(b ::' b)}"

The keyword `constdefs` introduces a constant definition. The definition and use of Pair_Rep is for convenience. There are "two names" ∗ and Prod.

See Product_Type.thy.

Example: Sums

An element of (α, β) `sum` is either Inl a ::' a or Inr b ::' b.

Consider type $\alpha \Rightarrow \beta \Rightarrow bool \Rightarrow bool$. We can regard $f : \alpha \Rightarrow \beta \Rightarrow bool \Rightarrow bool$ as a

representation of . . .	iff $f\,x\,y\,i$ is true for . . .
$Inl\ a$	$x = a$, y arbitrary, and $i = True$
$Inr\ b$	x arbitrary, $y = b$, and $i = False$.

Similar to pairs.

Isabelle Syntax for Sum Example

constdefs

```
  Inl_Rep  ::  ['a, 'a, 'b, bool] ⇒ bool
 "Inl_Rep ≡(λa. λx y p. x=a ∧p)"
  Inr_Rep  ::  ['b, 'a, 'b, bool] ⇒ bool
 "Inr_Rep ≡(λb. λx y p. y=b ∧¬p)"
```

typedef (Sum)

```
 ('a,'b)"+" =
   "{f. (∃ a. f = Inl_Rep(a ::'a)) ∨
        (∃ b. f = Inr_Rep(b ::'b))}"
```

See Sum_Type.thy.

Exercise: How would you define a type `even` based on `nat`?

Summary

- We have introduced a method to safely build up larger theories

Summary

- We have introduced a method to safely build up larger theories
- . . . and built sums and products

Summary

- We have introduced a method to safely build up larger theories
- . . . and built sums and products
- . . . and sets !
  (i.e. we have a method to overcome
  the problem of inconsistencies for
  the crucial problems !)

More Detailed Explanations

Axioms or Rules

Inside Isabelle, axioms are `thm`'s, and they may include Isabelle's metalevel implication $\Longrightarrow$. For this reason, it is not required to mention rules explicitly.

But speaking more generally about HOL, not just its Isabelle implementation, one should better say "rules" here, i.e., objects with a horizontal line and zero or more formulas above the line and one formula below the line.

Provable Formulas

The provable formulas are terms of type $bool$ derivable using the inference rules of HOL and the empty assumption list. We write $Th(T)$ for the derivable formulas of a theory T.

Closed Terms

A term is closed or ground if it does not contain any free variables.

Definition of $True$ Is Type-Closed

$True$ is defined as $\lambda x :: bool.\, x = \lambda x.\, x$ and not $\lambda x :: \alpha.x = \lambda x.\, x$. The definition must be type-closed.

Fixpoint Combinator

Given a function $f : \alpha \Rightarrow \alpha$, a fixpoint of f is a term t such that $f\,t = t$. Now Y is supposed to be a fixpoint combinator, i.e., for any function f, the term $Y\,f$ should be a fixpoint of f. This is what the rule

$$\frac{}{\forall f :: \alpha \Rightarrow \alpha . Y\,f = f\,(Y\,f)}\ \text{fix}$$

says. Consider the example $f \equiv \neg$. Then the axiom allows us to infer $Y(\neg) = \neg(Y(\neg))$, and it is easy to derive $False$ from this. This axiom is a standard example of a non-conservative extension of a theory.

This inconsistency is not surprising: Not every function has a fixpoint, so there cannot be a combinator returning a fixpoint of any function.

Nevertheless, fixpoints are important and must be realized in some way, as we will see later.

Side Conditions

By side conditions we mean

- E does not contain c and is closed;
- no subterm of E has a type containing a type variable that is not contained in the type of c;

in the definition.

The second condition also has a name: one says that the definition must be type-closed.

The notion of having a type is defined by the type assignment calculus. Since E is required to be closed, all variables occurring in E must be λ-bound, and so the type of those variables is given by the type superscripts.

Domains of Σ, Γ

The domain of Σ, denoted $dom(\Sigma)$, is $\{c \mid (c :: A) \in \Sigma$ for some $A\}$.

Likewise, the domain of Γ, denoted $dom(\Gamma)$, is
$\{x \mid (x :: A) \in \Gamma$ for some $A\}$.

Note the slight abuse of notation.

constdefs

In Isabelle theory files, `consts` is the keyword preceding a sequence of constant declarations (i.e., this is where the Σ is defined), and `defs` is the keyword preceding the constant definitions defining these constants (i.e., this is where the A is defined.

`constdefs` combines the two, i.e. it allows for a sequence of both constant declarations and definitions, and the theorem identifier c_def is generated automatically. E.g.

constdefs

```
 id  :: "'a ⇒ 'a"
"id ≡λ x. x"
```

will bind `id_def` to $id \equiv \lambda x.x$.

S

Here, S is any "predicate", i.e., term of type $R \Rightarrow bool$, not necessarily a constant.

Fresh T

The type constructor T must not occur in χ.

What Is T?

We use the letter χ to denote the set of type constructors (where the arity and fixity is indicated in some way). So since $T \in \chi'$, we have that T should be a type constructor. However, we abuse notation and also use T for the type obtained by applying the type constructor T to a vector of different type variables (as many as T requires).

$\uplus$

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.

What Are Abs_T and Rep_T?

Of course we are giving a schematic definition here, so any letters we use are meta-notation.

Notice that Abs_T and Rep_T stand for new constants. For any new type T to be defined, two such constants must be added to the signature to provide a generic way of obtaining terms of the new type. Since the new type is isomorphic to the "subset" S, whose members are of type R, one can say that Abs_T and Rep_T provide a type conversion between (the subset S of) R and T.

So we have a new type T, and we can obtain members of the new type by applying Abs_T to a term t of type R for which $S\,t$ holds.

Isomorphism

The formulas

$$\forall x. Abs_T(Rep_T\, x) = x$$
$$\forall x. S\, x \Rightarrow Rep_T(Abs_T\, x) = x$$

state that the "set" S and the new type T are isomorphic. Note that Abs_T should not be applied to a term not in "set" S. Therefore we have the premise $S\, x$ in the above equation.

Note also that S could be the "trivial filter" $\lambda x. True$. In this case, Abs_T and Rep_T would provide an isomorphism between the entire type R and the new type T.

Proof Obligation

We have said previously that S should be a non-empty "subset" of T. Therefore it must be proven that $\exists x.\, S\, x$. This is related to the semantics.

Whenever a type definition is introduced in Isabelle, the proof obligation must be shown inside Isabelle/HOL. Isabelle provides the `typedef` syntax for type definitions, as we will see later.

Inhabitation in the set Example

We have $S \equiv \lambda x :: \alpha \Rightarrow bool. True$, and so in $(\exists x. Sx)$, the variable x has type $\alpha \Rightarrow bool$. The proposition $(\exists x. Sx)$ is true since the type $\alpha \Rightarrow bool$ is inhabited, e.g. by the term $\lambda x :: \alpha. True$ or $\lambda x :: \alpha. False$.

Beware of a confusion: This does not mean that the new type $\alpha\, set$, 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 $Abs_{set}\ (\lambda x. False)$.

Recall a previous argument for the importance of inhabitation.

Trivial S

We said that in the general formalism for defining a new type, there is a term S of type $R \Rightarrow bool$ that defines a "subset" of a type R. In other words, it filters some terms from type R. 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 $\alpha\, set$. 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 $\alpha\, set$. This is not the case!

Rather, S is $\lambda x.True$ and hence trivial in this example. Moreover, in the example, R is $\alpha \Rightarrow bool$, and any term r of type R defines a set whose elements are of type α; $Abs_{set}\ r$ is that set.

$Collect$

We have seen $Collect$ before in the theory file exercise_03 (naïve set theory).

$Collect\, f$ is the set whose characteristic function is f. The usual concrete syntax is $\{x \mid f\, x\}$. The construct is called set comprehension.

Note also that $Collect$ is the same as Abs_{set} here, so there is no need to have them as separate constants, and for this reason Isabelle theory file Set.thy only provides $Collect$.

The $\in$-Sign

We define

$$x \in A \;\; = \;\; (Rep_{set}\; A)\; x$$

Since Rep_{set} has type $\alpha\, set \Rightarrow (\alpha \Rightarrow bool)$, this means that x is of type α and A is of type $(\alpha \Rightarrow bool)$. Therefore $\in$ is of type $\alpha \Rightarrow (\alpha\, set) \Rightarrow bool$ (but written infix).

In the the Isabelle theory Set.thy, you will indeed find that the constant op : (Isabelle syntax for $\in$) has type $[\alpha, \alpha\, set] \Rightarrow bool$. However, you will not find anything directly corresponding to Rep_{set}.

One can see that this setup is equivalent to the one we have here (which was presented like that for the sake of generality). There are two axioms in Set.thy:

axioms

mem_Collect_eq [iff]: "(a : {x. P(x)}) = P(a)"

Collect_mem_eq [simp]: "{x. x:A} = A"

These axioms can be translated into definitions as follows:

$$a \in \{x \mid P\,x\} = P\,a \rightsquigarrow$$
$$a \in (Collect\,P) = P\,a \rightsquigarrow$$
$$a \in (Abs_{set}\,P) = P\,a \rightsquigarrow$$
$$Rep_{set}(Abs_{set}\,P)\,a = P\,a \rightsquigarrow Rep_{set}(Abs_{set}\,P) = P$$

The last step uses extensionality.

Now the second one:

$$\{x \mid x \in A\} = A \rightsquigarrow$$
$$\{x \mid (Rep_{set}A)\,x\} = A \rightsquigarrow$$
$$Collect(Rep_{set}A) = A$$

Ignoring some universal quantifications (these are implicit in Isabelle),

these are the isomorphy axioms for set.

Consistent Set Theory

Typed set theory is a conservative extension of HOL and hence consistent.

Recall the problems with untyped set theory.

"Exactly one" Term

When we say that there is "exactly one" f, this is meant modulo equality in HOL. This means that e.g. $\lambda x :: \alpha\, y :: \beta . y = b \wedge x = a$ is also such a term since $(\lambda x :: \alpha\, y :: \beta . x = a \wedge y = b) = (\lambda x :: \alpha y :: \beta .\, y = b \wedge x = a)$ is derivable in HOL.

$Rep_\times$

$Rep_\times$ would be the generic name for one of the two isomorphism-defining functions.

Since $Rep_\times$ can not be represented directly for lexical reasons, type definitions in Isabelle provide 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.

Iteration of λ's

We write $\lambda a :: \alpha\ b :: \beta.\ \lambda x :: \alpha\ y :: \beta.\ x = a \wedge y = b$ rather than $\lambda a :: \alpha\ b :: \beta\ x :: \alpha\ y :: \beta . x = a \wedge y = b$ to emphasize the idea that one first applies $Pair_Rep$ to a and b, and the result is a function representing a pair, wich can then be applied to x and y.

Sum Types

Idea of sum or union type: t is in the sum of τ and σ if t is either in τ or in σ. To do this formally in our type system, and also in the type system of functional programming languages like ML, t must be wrapped to signal if it is of type τ or of type σ.

For example, in ML one could define

$$\texttt{datatype}\ (\alpha, \beta)\ \texttt{sum} = \mathit{Inl}\ \alpha \mid \mathit{Inr}\ \beta$$

So an element of (α, β) sum is either $\mathit{Inl}\ a$ where $a :: \alpha$ or $\mathit{Inr}\ b$ where $b :: \beta$.

Defining `even`

Suppose we have a type `nat` and a constant $+$ with the expected meaning. We want to define a type `even` of even numbers. What is an even number?

Defining `even`

Suppose we have a type `nat` and a constant $+$ with the expected meaning. We want to define a type `even` of even numbers. What is an even number?

The following choice of S is adequate:

$$S \equiv \lambda x.\, \exists n.\, x = n + n$$

Using the Isabelle scheme, this would be

typedef (Even)
  even = "{x. ∃ y.x=y+y}"

We could then go on by defining an operation `PLUS` on `even`, say as follows:

constdefs

PLUS::[even,even] $\longrightarrow$ even (**infixl** 56)
PLUS_def "op PLUS $\equiv\lambda$xy. Abs_Even(Rep_Even(x)+Rep_Even(x))"

Note that we chose to use names `even` and `Even`, but we could have used the same name twice as well.

HOL: Basic Library

Burkhart Wolff

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

Discoveries in mathematics:

Sets: The Basis of Principia Mathematica

Discoveries in mathematics:

- 17th century: geometry can be reduced to numbers (Descartes, Leibniz, . . .)

Sets: The Basis of Principia Mathematica

Discoveries in mathematics:

- 17th century: geometry can be reduced to numbers (Descartes, Leibniz, . . .)
- 19th century: numbers can be reduced to sets (Cantor, Peano, Frege [Fre93, Fre03], . . .)

Sets: The Basis of Principia Mathematica

Discoveries in mathematics:

- 17th century: geometry can be reduced to numbers (Descartes, Leibniz, . . .)
- 19th century: numbers can be reduced to sets (Cantor, Peano, Frege [Fre93, Fre03], . . .)
- 20th century: sets can be represented in logics (Zermelo/Fränkel, Russel/Whitehead [WR25], Gödel/Bernays, . . .)

Sets: The Basis of Principia Mathematica

Discoveries in mathematics:

- 17th century: geometry can be reduced to numbers (Descartes, Leibniz, . . .)
- 19th century: numbers can be reduced to sets (Cantor, Peano, Frege [Fre93, Fre03], . . .)
- 20th century: sets can be represented in logics (Zermelo/Fränkel, Russel/Whitehead [WR25], Gödel/Bernays, . . .)

As a collection of formalized theories, we call this the Principia Mathematica Structure [WR25].

Interesting: The libraries of modern theorem provers follow this structure . . .

The Roadmap

- Orders
- Sets
- Functions
- (Least) fixpoints and induction
- (Well-founded) recursion
- Arithmetic
- Datatypes

Orders

The Roadmap

We are looking at how the different parts of mathematics are encoded in the Isabelle/HOL library.

- Orders
- Sets
- Functions
- (Least) fixpoints and induction
- (Well-founded) recursion
- Arithmetic
- Datatypes

The Roadmap

We are looking at how the different parts of mathematics are encoded in the Isabelle/HOL library.

- Orders
- Sets
- Functions
- (Least) fixpoints and induction
- (Well-founded) recursion
- Arithmetic
- Datatypes

Three Order Classes

We first define a syntactic class `ord`. It is the class of types for which symbols $<$ and $\leq$ exist.

Three Order Classes

We first define a syntactic class `ord`. It is the class of types for which symbols $<$ and $\leq$ exist.

We then define two axiomatic classes order and linorder for which $<$ and $\leq$ are required to have certain properties, that of being a partial order, or a linear order, resp.

Orders (in HOL.thy)

axclass

ord < type

consts

"op <" :: ['a::ord, 'a] ⇒ bool

"op ≤" :: ['a::ord, 'a] ⇒ bool

constdefs

min :: "['a::ord, 'a] ⇒ 'a"

"min a b ≡ (if a ≤ b then a else b)"

max :: "['a::ord, 'a] ⇒ 'a"

"max a b ≡ (if a ≤ b then b else a)"

Recall **constdefs** syntax and note two uses of <.

Orders in HOL.thy (Cont.)

axclass order < ord

order_refl "x ≤ x"

order_trans "⟦x ≤ y; y ≤ z⟧ ⟹ x ≤ z"

order_antisym "⟦x ≤ y; y ≤ x⟧ ⟹ x = y"

order_less_le "x < y = (x ≤ y ∧ x ~= y)"

axclass linorder < order

linorder_linear "x ≤ y ∨ y ≤ x"

Least Elements

In HOL.thy, least elements used to be defined as:

```
Least :: "('a::ord ⇒ bool) ⇒ 'a"
Least_def "Least P ≡THE x. P(x) ∧
                    (∀ y. P(y) ⟹x ≤y)"
```

Monotonicity

In HOL.thy, monotonicity used to be defined as:

$$\begin{array}{lll} \text{mono} & :: & [\text{'a} :: \text{ord} \Rightarrow \text{'b} :: \text{ord}] \Rightarrow \text{bool} \\ \text{mono_def} & & \text{''mono(f)} \equiv \\ & & (\forall A\ B.\ A \leq B \Longrightarrow f(A) \leq f(B)) \end{array}$$

Some Theorems about Orders

monoI	$(\bigwedge AB.A \leq B \Longrightarrow f\,A \leq f\,B)$ $\Longrightarrow mono\,f$
monoD	$[\![mono\,f; A \leq B]\!] \Longrightarrow f\,A \leq f\,B$
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	$min\,x\,x = x$
le_min_iff_conj	$(z \leq min\,x\,y) = (z \leq x \wedge z \leq y)$

Discussion of Orders

Type classes are a structuring mechanism in Isabelle:

- Syntactic classes (e.g. $t :: \alpha :: ord$ as in Haskell [HHPW96]): merely a mechanism to structure visibility of operations.

Discussion of Orders

Type classes are a structuring mechanism in Isabelle:

- Syntactic classes (e.g. $t :: \alpha :: ord$ as in Haskell [HHPW96]): merely a mechanism to structure visibility of operations.
- Axiomatic classes (e.g. $t :: \alpha :: order$): a mechanism for structuring semantic knowledge in types (foundation to be discussed later).

Sets

The Roadmap

We are still looking at how the different parts of mathematics are encoded in the Isabelle/HOL library.

- Orders
- Sets
- Functions
- (Least) fixpoints and induction
- (Well-founded) recursion
- Arithmetic
- Datatypes

The Roadmap

We are still looking at how the different parts of mathematics are encoded in the Isabelle/HOL library.

- Orders
- Sets
- Functions
- (Least) fixpoints and induction
- (Well-founded) recursion
- Arithmetic
- Datatypes

Set.thy

theory Set = HOL:

typedecl 'a set

instance set :: (type) ord ..

consts

```
 "{}"      :: 'a set ("{}")
 UNIV      :: 'a set
 Collect   :: ('a ⇒ bool) ⇒ 'a set
 "op :"    :: "'a ⇒ 'a set ⇒ bool"
```

Note that `Collect` and ":" (alias: $\in$) correspond to Abs_{set} and Rep_{set}.

Set.thy: More Constant Declarations

insert	::	['a, 'a set] ⇒ 'a set
∪, ∩	::	['a set, 'a set] ⇒ 'a set
Ball, Bex	::	['a set, 'a ⇒ bool] ⇒ bool
UNION, INTER	::	['a set, 'a ⇒ 'b set] ⇒ 'b set
⋃, ⋂	::	(('a set) set) ⇒ 'a set

There is the equivalent syntax:

$\{x, y, z\}$	for	$\texttt{insert}\ x\ (\texttt{insert}\ y\ (\texttt{insert}\ z\ \{\}))$
$\forall\, x : A.\, Sx$	for	$\texttt{Ball}\ A\ S$
$\exists\, x : A.\, Sx$	for	$\texttt{Bex}\ A\ S$
$\bigcup_{x \in A} .\, S\, x$	for	$\texttt{UNION}\ A\ S$
$\bigcap_{x \in A} .\, S\, x$	for	$\texttt{INTER}\ A\ S$

Set.thy: Constant Definitions

empty_def: "$\{\} \equiv \{x.\ \text{False}\}$"

UNIV_def: "$\text{UNIV} \equiv \{x.\ \text{True}\}$"

Un_def: "$A \cup B \equiv \{x.\ x \in A \vee x \in B\}$"

Int_def : "$A \cap B \equiv \{x.\ x \in A \wedge x \in B\}$"

insert_def : "$\textit{insert}\ a\ B \equiv \{x.\ x = a\} \cup B$"

Ball_def : "$\text{Ball}\ A\ P \equiv \forall x.\ x \in A \longrightarrow P(x)$"

Bex_def: "$\text{Bex}\ A\ P \equiv \exists x.\ x \in A \wedge P(x)$"

Set.thy: Constant Definitions (2)

subset_def : "A $\leq$B $\equiv\forall$ x$\in$A. x$\in$B"

Compl_def: "$-$ A $\equiv\{$x. $\neg$x$\in$A$\}$"

set_diff_def : "A $-$ B $\equiv\{$x. x$\in$A $\wedge\neg$x$\in$B$\}$"

UNION_def: "UNION A B $\equiv\{$y. $\exists$x$\in$A. y$\in$ B(x)$\}$"

INTER_def: "INTER A B $\equiv\{$y. $\forall$x$\in$A. y$\in$B(x)$\}$"

Set.thy: Constant Definitions (2)

subset_def : "A $\leq$B $\equiv\forall$ x$\in$A. x$\in$B"

Compl_def: "$-$ A $\equiv\{$x. $\neg$x$\in$A$\}$"

set_diff_def : "A $-$ B $\equiv\{$x. x$\in$A $\wedge\neg$x$\in$B$\}$"

UNION_def: "UNION A B $\equiv\{$y. $\exists$x$\in$A. y$\in$ B(x)$\}$"

INTER_def: "INTER A B $\equiv\{$y. $\forall$x$\in$A. y$\in$B(x)$\}$"

Note use of $\leq$ instead of $\subseteq$!

Set.thy: Constant Definitions (3)

Union_def: "$\bigcup$ S $\equiv$($\bigcup$ x$\in$S. x)"

Inter_def : "$\bigcap$ S $\equiv$(INT x$\in$S. x)"

Pow_def: "Pow A $\equiv$$\{$B. B $\leq$A$\}$"

image_def: "f`A $\equiv$$\{$y. $\exists$ x$\in$A. y = f(x)$\}$"

Some Theorems in Set.thy

`CollectI`	$P\,a \Longrightarrow a \in \{x.P\,x\}$
`CollectD`	$a \in \{x.P\,x\} \Longrightarrow P\,a$
`set_ext`	$(\bigwedge x.(x \in A) = (x \in B)) \Longrightarrow A = B$
`subsetI`	$(\bigwedge x.x \in A \Longrightarrow x \in B) \Longrightarrow A \subseteq B$
`eqset_imp_iff`	$A = B \Longrightarrow (x \in A) = (x \in B)$
`UNIV_I`	$x \in \mathtt{UNIV}$
`subset_UNIV`	$A \subseteq \mathtt{UNIV}$
`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 in Set.thy

`insert_iff`	$(a \in insert\, b\, A) = (a = b \vee a \in A)$
`image_Un`	$f`(A \cup B) = f`A \cup f`B$
`Inter_lower`	$B \in A \Longrightarrow \bigcap A \subseteq B$
`Inter_greatest`	$(\bigwedge X.X \in A \Longrightarrow C \subseteq X) \Longrightarrow C \subseteq \bigcap A$

Discussion of Sets

Rich and powerful set theory available in HOL:

- No problems with consistency
- Weaker than ZF (since typed set-theory:) there is no "union of sets"; but: complement-closed
- Good mechanical support for many set tautologies (both for classical reasoning as well as simplification)
- Powerful basis for many problems in modeling.

Functions

The Roadmap

We are still looking at how the different parts of mathematics are encoded in the Isabelle/HOL library.

- Orders
- Sets
- Functions
- (Least) fixpoints and induction
- (Well-founded) recursion
- Arithmetic
- Datatypes

The Roadmap

We are still looking at how the different parts of mathematics are encoded in the Isabelle/HOL library.

- Orders
- Sets
- Functions
- (Least) fixpoints and induction
- (Well-founded) recursion
- Arithmetic
- Datatypes

Fun.thy

The theory Fun.thy defines some important notions on functions, such as concatenation, the identity function, the image of a function, etc.

We look at it briefly.

Two Extracts from Fun.thy

Composition and the identity function:

constdefs

id :: "'a ⇒ 'a"
"id ≡ λ x. x"

comp :: "['b ⇒ 'c, 'a ⇒ 'b, 'a] ⇒ 'c"
"f o g ≡ λ x. f(g(x))"

There are also definitions for function update, function override and theorems for concepts such as injectivity, surjectivity and bijectivity.

Compare the syntax for `constdefs`.

Instantiating an Axiomatic Class

Sets are partial orders: `set` is an instance of the axiomatic class `order`.

instance set :: (type) order
  apply (auto) **done**

- Axiomatic classes result in proof obligations.
- These are discharged whenever instance is stated.
- Type-checking has access to the established properties.

Conclusion of Orders, Sets, and Functions

- conservative extensions can be used to build consistent libraries.

Conclusion of Orders, Sets, and Functions

- conservative extensions can be used to build consistent libraries.
- Sets as one important package of Isabelle/HOL library:
  - Set theory is typed, but very rich and powerfully supported.

Conclusion of Orders, Sets, and Functions

- conservative extensions can be used to build consistent libraries.
- Sets as one important package of Isabelle/HOL library:
  - Set theory is typed, but very rich and powerfully supported.
  - Sets are instance of `ord` and `order` type class, demonstrates type classes as structuring mechanism in Isabelle.

Conclusion of Orders, Sets, and Functions

- conservative extensions can be used to build consistent libraries.
- Sets as one important package of Isabelle/HOL library:
  - Set theory is typed, but very rich and powerfully supported.
  - Sets are instance of `ord` and `order` type class, demonstrates type classes as structuring mechanism in Isabelle.

More Detailed Explanations

Different uses of $<$

Note: the $<$ may occur in different lexical categories, e.g.:

axclass order $<$ ord

in the theory file states that `order` is a subclass of `ord`.

Compare to the declaration

"op $<$" :: ['a :: ord, 'a] $\Rightarrow$ bool ("(_ $<$ _)" [50, 51] 50)

where a constant $<$ with a certain type is introduced.

Semantic Classes for Semantic Knowledge

The Isabelle type system records for any type variable what class constraints there are for this type variable. These class constraints may arise from the types of the constants used in an expression, or they may be given explicitly by the user in a goal. E.g. one might type

lemma "(x::'a::order)<y ⟹x≤y";

to specify that x must be of a type in the type class `order`.

The axioms of an axiomatic class can only be applied if any constant declared in the axiomatic class (or a syntactic superclass) is applied to arguments of a type in the axiomatic class. E.g. order_refl can only be used to prove y ≤y if the type of y is in the type class order.

In this sense the type information (y is of type in class order) is semantic knowledge (y ≤y holds).

$\leq$ instead of $\subseteq$

Sets are an instance of the type class ord, where the generic constant $\leq$ is the subset relation in this particular case.

In fact, the subset relation is reflexive, transitive and anti-symmetric, and so sets are an instance of the axiomatic class order. This is non-obvious and must be proven as part of the **instance** statement.

Union of Arbitrary Sets?

In typed set theory (what we have here in HOL), it is not possible to form the union of two sets of different type. This is in contrast to ZF.

Typed Sets Are Complement-Closed

The complement of a typed set A, i.e.

$$\{x \mid x \notin A\}$$

is again a set, whose type is the same as the type of A. In ZF, the complement construction is not generally allowed since it opens the door to Russell's Paradox.

Proof Obligations

To claim that a type is an instance of an axiomatic class, it has to be proven that the axioms (in the case pf order: order_refl , order_trans , order_antisym, and order_less_le are indeed fulfilled by that type.

Discharge Obligations

The Isabelle mechanism is such that the line

```
instance set :: (type) order
  apply(auto) done
```

instructs Isabelle to prove the axioms using the previously proven theorems subset_refl , subset_trans , subset_antisym, and psubset_eq.

Higher-Order Logic: Fixpoints and Inductive Definitions

Jan-Georg Smaus and Burkhart Wolff

Recursion in HOL

Current stage of our course:

- On the basis of conservative extensions, set theory can be built safely.

Recursion in HOL

Current stage of our course:

- On the basis of conservative extensions, set theory can be built safely.
- But: our mathematical language is still quite small. Conservative extensions rule out recursive definitions, be it recursive programs or recursive set equations, . . .).

Recursion in HOL

Current stage of our course:

- On the basis of conservative extensions, set theory can be built safely.
- But: our mathematical language is still quite small. Conservative extensions rule out recursive definitions, be it recursive programs or recursive set equations, . . .).

How can we benefit from set theory to introduce some form of recursion?

Recursion and General Fixpoints

Naïve Approach: One could axiomatize fixpoint combinator Y as

$$\frac{}{Y = \lambda F.F(Y\,F)}\ \text{fix}$$

This axiom is not a constant definition.

Then we could easily derive

$$\forall F^{\alpha \Rightarrow \alpha}.Y\ F = F\,(Y\ F).$$

Recursion and General Fixpoints

Naïve Approach: One could axiomatize fixpoint combinator Y as

$$\frac{}{Y = \lambda F.F(YF)}\ \text{fix}$$

This axiom is not a constant definition.

Then we could easily derive

$$\forall F^{\alpha \Rightarrow \alpha}.Y\ F = F\,(Y\ F).$$

- Why are we interested in Y?
- What's the problem with such a definition?

Why Are We Interested in Y?

First, why are we interested in recursion (solutions to recursive equations)?

Why Are We Interested in Y?

First, why are we interested in recursion (solutions to recursive equations)?

- Recursively defined functions are solutions of such equations (example: fac).
- Inductively defined sets are solutions of such equations (example: $Fin\ A$, all finite subsets of A).

Why Are We Interested in Y?

First, why are we interested in recursion (solutions to recursive equations)?

- Recursively defined functions are solutions of such equations (example: fac).
- Inductively defined sets are solutions of such equations (example: $Fin\ A$, all finite subsets of A).

We are interested in Y because it is the mother of all recursions. With Y, recursive axioms can be converted into constant definitions.

What's the Problem with such an Axiom?

- an axiom would lead to inconsistency. (Consider $\neg$).

What's the Problem with such an Axiom?

- an axiom would lead to inconsistency. (Consider $\neg$).
- This is not surprising because not all functions have a fixpoint.

What's the Problem with such an Axiom?

- an axiom would lead to inconsistency. (Consider $\neg$).
- This is not surprising because not all functions have a fixpoint.
- Therefore we only consider special forms of fixpoint combibinators, the two approaches

What's the Problem with such an Axiom?

- an axiom would lead to inconsistency. (Consider $\neg$).
- This is not surprising because not all functions have a fixpoint.
- Therefore we only consider special forms of fixpoint combibinators, the two approaches
  - Least fixpoints
  - well-founded orderings.

What's the Problem with such an Axiom?

- an axiom would lead to inconsistency. (Consider $\neg$).
- This is not surprising because not all functions have a fixpoint.
- Therefore we only consider special forms of fixpoint combibinators, the two approaches
  - Least fixpoints
  - well-founded orderings.

The following section is devoted to least fixpoints, which happen to be closely related to inductive definitions.

More Detailed Explanations

Axiom is not a Definition

The axiom

$$Y = \lambda F.F(YF)$$

is not a constant definition, since Y occurs again on the right-hand side.

$$\forall F^{\alpha \Rightarrow \alpha}. Y\ F = F\ (Y\ F)$$

In words, this says that $Y\ F$ is a fixpoint of F.

Recursive Equation

By a recursive equation, we mean an equation of the form

$$f = e$$

where f occurs in e. Such an equation does not satisfy the requirements of a constant definition.

Converting Recursive Axioms

Any recursive function can be defined by an expression (functional) which is not itself recursive, but instead relies on the recursive equation defining Y.

Consider fac or $Fin\ A$ as an example.

Defining fac

In the following explanations, any constants like 1 or $+$ or **if-then-else** are intended to have their usual meaning.

A fixpoint combinator is a function Y that returns a fixpoint of a function F, i.e., Y must fulfill the equation $YF = F(YF)$. Doing λ-abstraction over F on both sides and η-conversion (backwards) on the left-hand side, we have

$$Y = \lambda F.F(YF)$$

This is a recursive equation. We will now demonstrate how a definition of a function fac (factorial) using a recursive equation can be transformed to a definition that uses Y instead of using recursion directly.

In a functional programming language we might define

$$fac\ n = (\mathbf{if}\ n = 0\ \mathbf{then}\ 1\ \mathbf{else}\ n * fac\ (n-1)).$$

We now massage this equation a bit. Doing λ-abstraction on both sides we get

$$\lambda n.\, fac\ n = (\lambda n.\ \mathbf{if}\ n = 0\ \mathbf{then}\ 1\ \mathbf{else}\ n * fac(n-1))$$

which is the η-conversion of

$$fac = (\lambda n.\ \mathbf{if}\ n = 0\ \mathbf{then}\ 1\ \mathbf{else}\ n * fac(n-1))$$

which in turn is a β-reduction of

$$fac = (\underline{(\lambda f.\, \lambda n.\ \mathbf{if}\ n = 0\ \mathbf{then}\ 1\ \mathbf{else}\ n * f(n-1))}\ fac) \tag{3}$$

We are looking for a solution to (3). We abbreviate the underlined expression by Fac. We claim $fac = Y\ Fac$, i.e., it is a solution to (3).

Simply replacing fac with $Y\ Fac$ in (3) we get

$$Y\ Fac = Fac\ (Y\ Fac)$$

which holds by the definition of Y.

Thus we see that a recursive definition of a function can be transformed so that the function is the fixpoint of an appropriate functional (a function taking a function as argument).

Defining $Fin\ A$

We want to define a function Fin such that $Fin\ A$ is the set of all finite subsets of A.

How do you construct the set of all finite subsets of A? The following pseudo-code suggests what you have to do:

$$
\begin{array}{l}
S := \{\{\}\}; \\
\textbf{forever do} \\
\quad \textbf{foreach } a \in A \textbf{ do} \\
\quad\quad \textbf{foreach } B \in S \textbf{ do} \\
\quad\quad\quad \textbf{add } (\{a\} \cup B) \textbf{ to } S \\
\textbf{od od od}
\end{array}
$$

This means that you have to add new sets forever (however, when you actually do this construction for a finite set A, it will indeed reach a

fixpoint, i.e., adding new sets won't change anything).

Generally (even if A is infinite), $Fin\ A$ is a set such that adding new sets as suggested by the pseudo-code won't change anything. Written as recursive equation:

$$Fin\ A = \{\{\}\} \cup \bigcup x \in A.((\texttt{insert}\ x)\ `\ Fin\ A)$$

Recall that ‘ is nice syntax for $image$, defined in `Set.thy`.

The above is a β-reduction of

$$Fin\ A = \underline{(\lambda X.\ \{\{\}\} \cup \bigcup x \in A.((\texttt{insert}\ x)\ `\ X))}\ (Fin\ A) \qquad (4)$$

We are looking for a solution to (4). We abbreviate the underlined expression by FA. We claim

$$Fin\ A = Y\ FA,$$

i.e., it is a solution to (4). Simply replacing $Fin\ A$ with $Y\ FA$ in (4) we get

$$Y\ FA = FA(Y\ FA),$$

which holds by the definition of Y.

You should compare this to what we said about fac. Note that in this example, there is no such thing as a recursive call to a "smaller" argument as in fac example.

Least Fixpoints

Jan-Georg Smaus and Burkhart Wolff

The Roadmap

We are still looking at how the different parts of mathematics are encoded in the Isabelle/HOL library.

- Orders
- Sets
- Functions
- (Least) fixpoints and induction
- (Well-founded) recursion
- Arithmetic
- Datatypes

The Roadmap

We are still looking at how the different parts of mathematics are encoded in the Isabelle/HOL library.

- Orders
- Sets
- Functions
- (Least) fixpoints and induction
- (Well-founded) recursion
- Arithmetic
- Datatypes

A Conservative Approach for Least Fixpoints

- Restriction: F is not of arbitrary type $(\alpha \Rightarrow \alpha)$ but of set type $(\alpha\ set \Rightarrow \alpha\ set)$.
- 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, and: this is not obvious (requires non-trivial proof)!

This leads us towards recursion and induction.

Lfp.thy

constdefs

lfp :: ['a set ⇒ 'a set] ⇒ 'a set

" lfp(f) == ⋂({u. f(u) ⊆u})"

Note: The definition of *lfp* is conservative.

Lfp.thy

constdefs

lfp :: ['a set ⇒ 'a set] ⇒ 'a set

" lfp(f) == ⋂({u. f(u) ⊆u})"

Note: The definition of *lfp* is conservative.

That's fine. But is it a fixpoint combinator?

Knaster-Tarski's Fixpoint Theorem

Theorem 5 (Knaster-Tarski):

If f is monotone, then $lfp\ f = f\ (lfp\ f)$.

The proof has four steps.

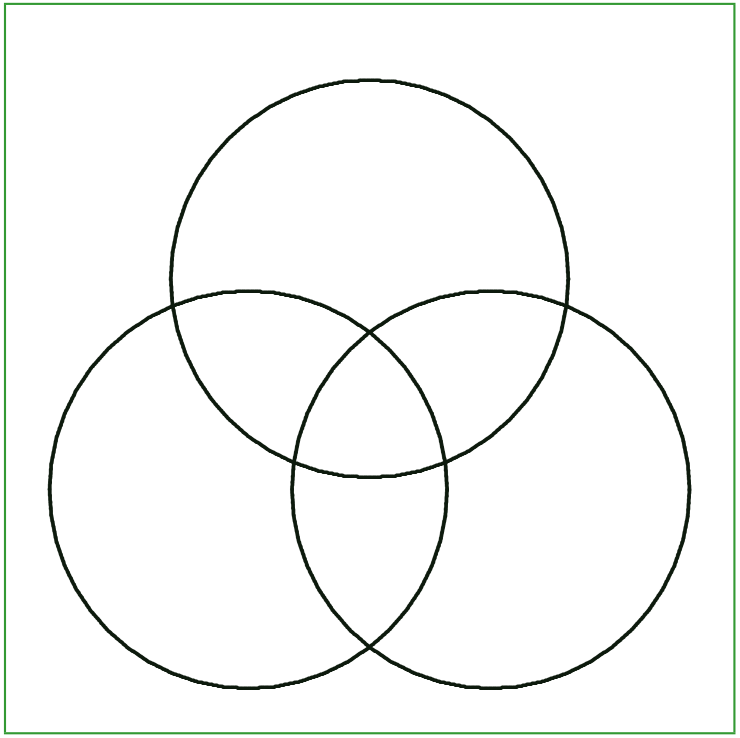
Knaster-Tarski's Fixpoint Theorem (1)

Claim 1 ("lfp lower bound"): If $f\ A \subseteq A$ then $lfp\ f \subseteq A$.

Knaster-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. The three circles denote the sets A for which $f\ A \subseteq A$.

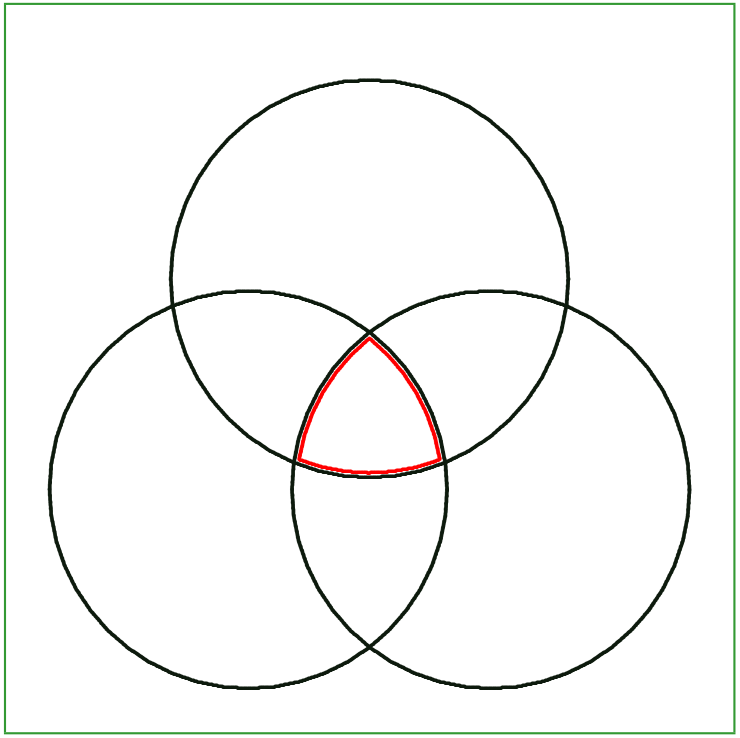


Knaster-Tarski's Fixpoint Theorem (1)

Claim 1 ("lfp lower bound"): If $f\ A \subseteq A$ then $lfp\ f \subseteq A$.

The box denotes "the set" α. The three circles denote the sets A for which $f\ A \subseteq A$.

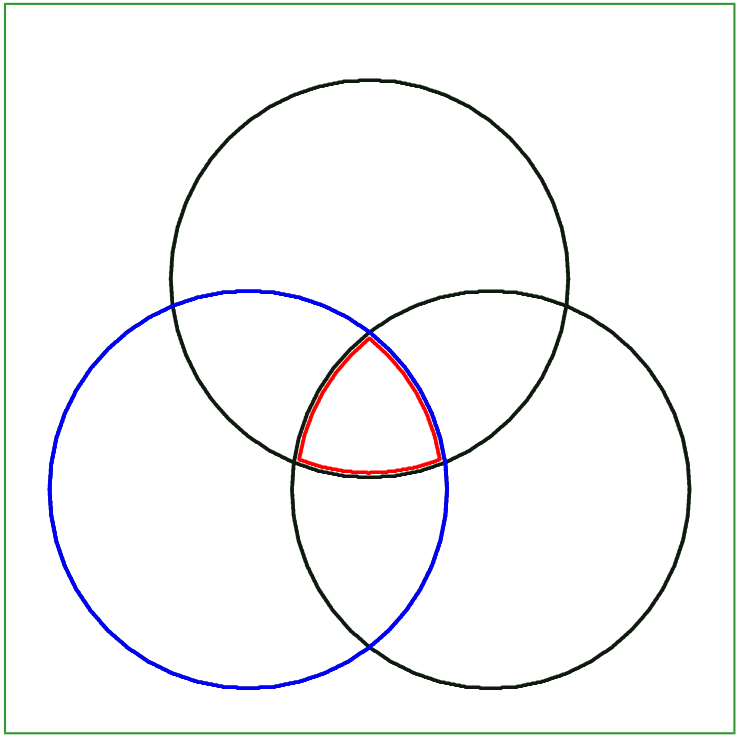
By Lfp.thy, $lfp\ f$ is the intersection.



Knaster-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. The three circles denote the sets A for which $f\ A \subseteq A$.
By Lfp.thy, $lfp\ f$ is the intersection.
Pick an A for which $f\ A \subseteq A$.
Clearly, $lfp\ f \subseteq A$.



Or as proof tree.

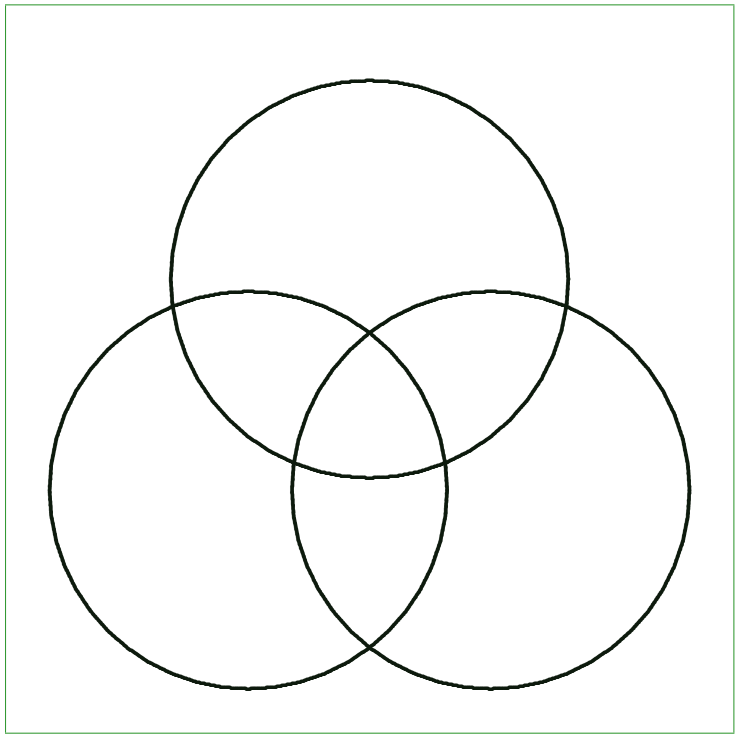
Knaster-Tarski's Fixpoint Theorem (2)

Claim 2 ("lfp greatest"): For all A, if
for all U, $f\ U \subseteq U$ implies $A \subseteq U$, then $A \subseteq lfp\ f$.

Knaster-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 $A \subseteq lfp\ f$.

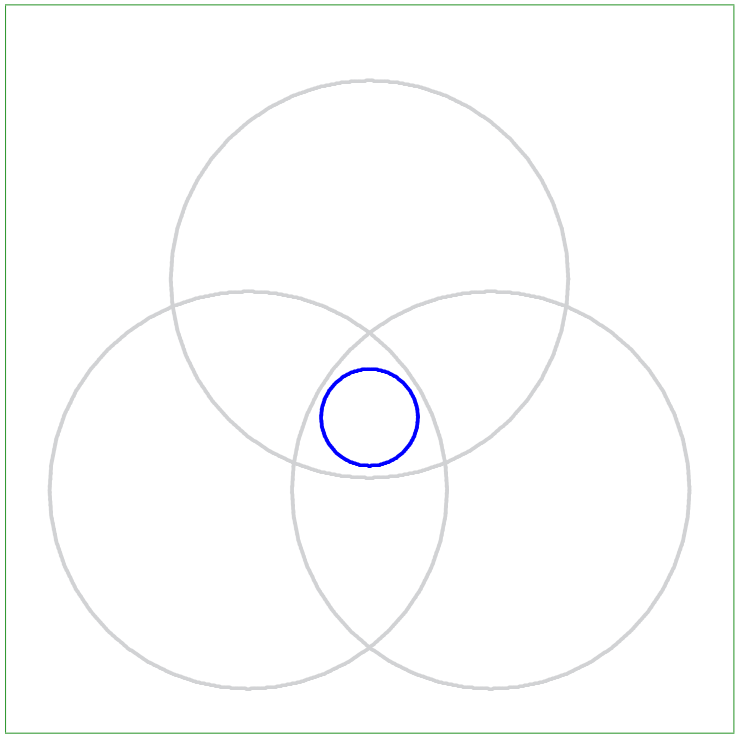
The three circles denote the sets U for which $f\ U \subseteq U$.



Knaster-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 $A \subseteq lfp\ f$.

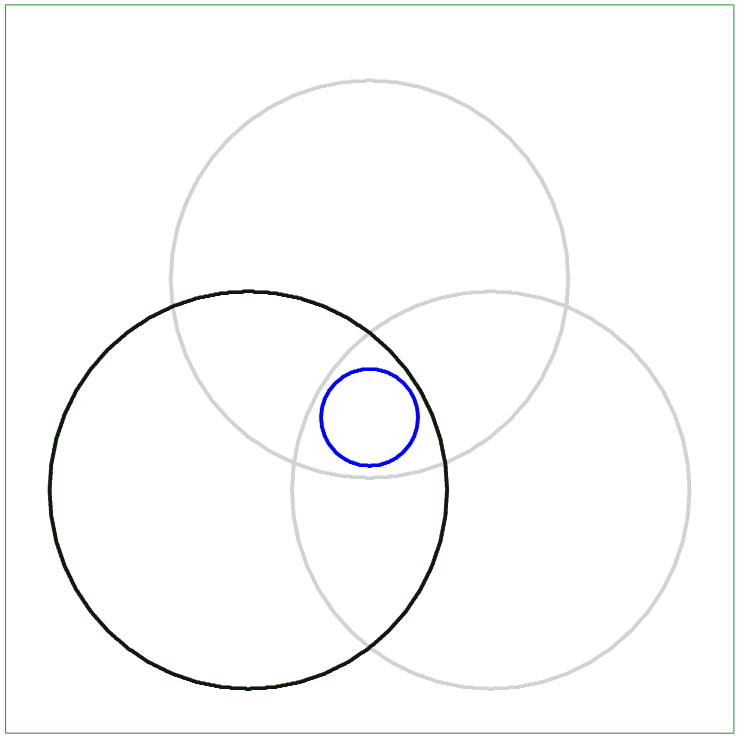
The three circles denote the sets U for which $f\ U \subseteq U$.
By <u>hypothesis</u>, $A \subseteq U$ for each U



Knaster-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 $A \subseteq lfp\ f$.

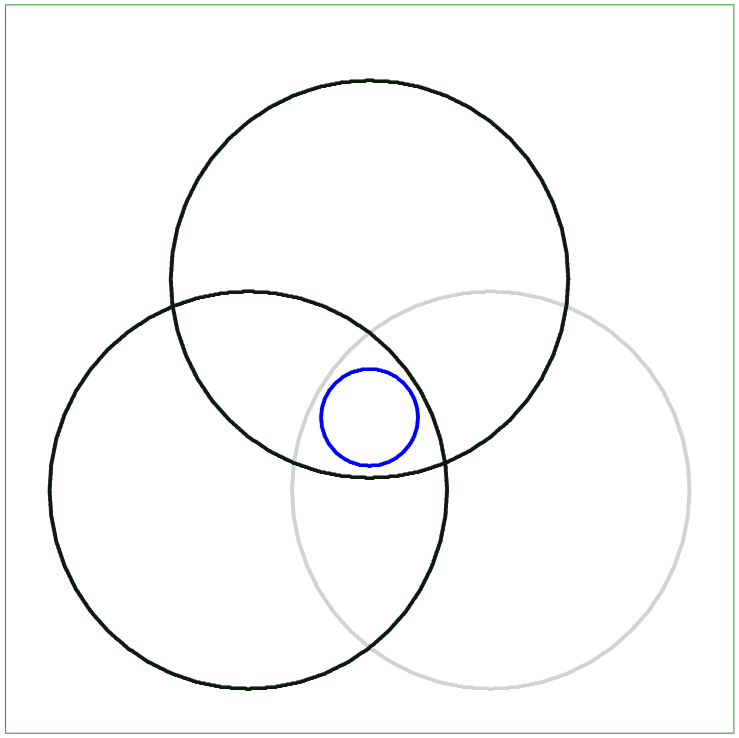
The three circles denote the sets U for which $f\ U \subseteq U$.
By hypothesis, $A \subseteq U$ for each U (1st,



Knaster-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 $A \subseteq lfp\ f$.

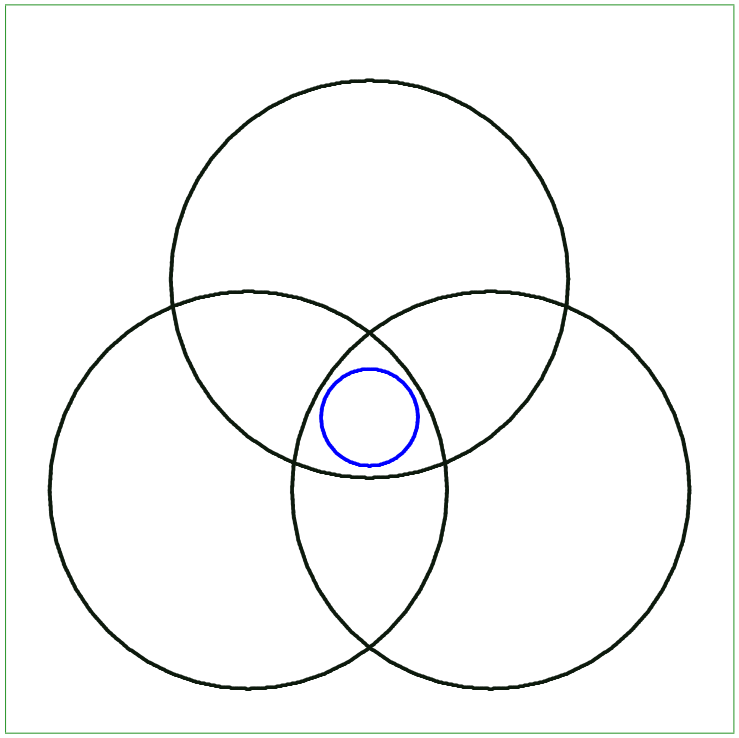
The three circles denote the sets U for which $f\ U \subseteq U$.
By hypothesis, $A \subseteq U$ for each U (1st, 2nd,



Knaster-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 $A \subseteq lfp\ f$.

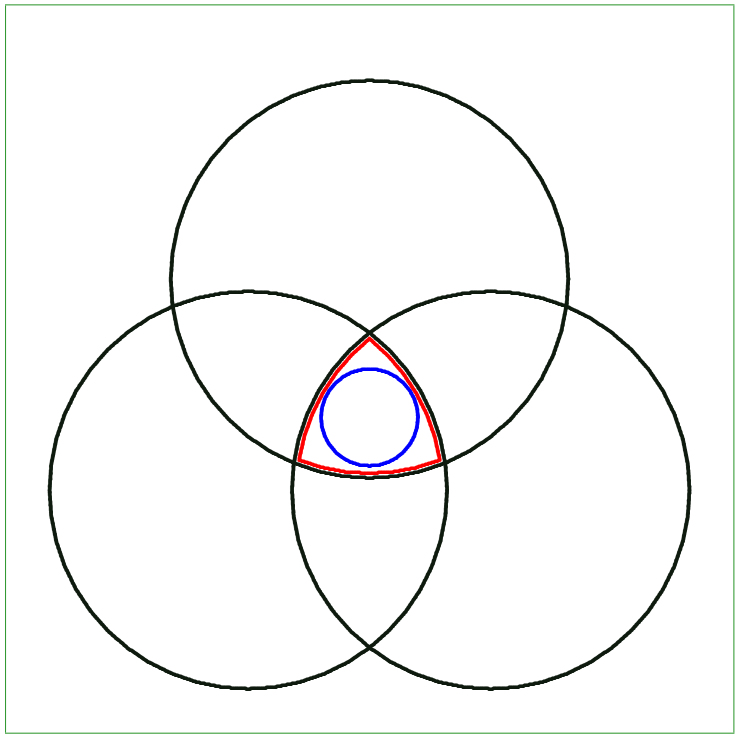
The three circles denote the sets U for which $f\ U \subseteq U$.
By hypothesis, $A \subseteq U$ for each U (1st, 2nd, 3rd . . .).



Knaster-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 $A \subseteq lfp\ f$.

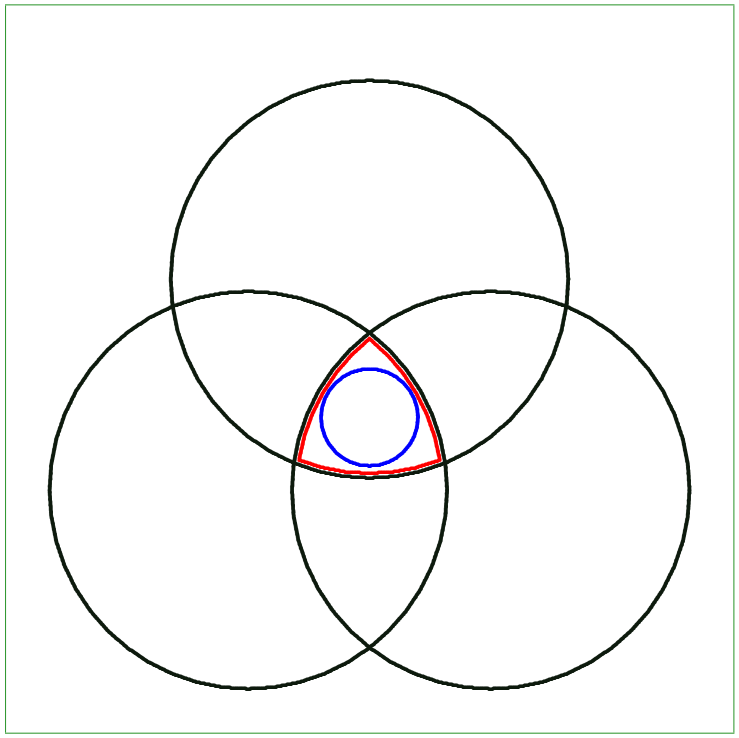
The three circles denote the sets U for which $f\ U \subseteq U$.
By hypothesis, $A \subseteq U$ for each U (1st, 2nd, 3rd . . .).
By definition, $lfp\ f$ is the intersection.



Knaster-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 $A \subseteq lfp\ f$.

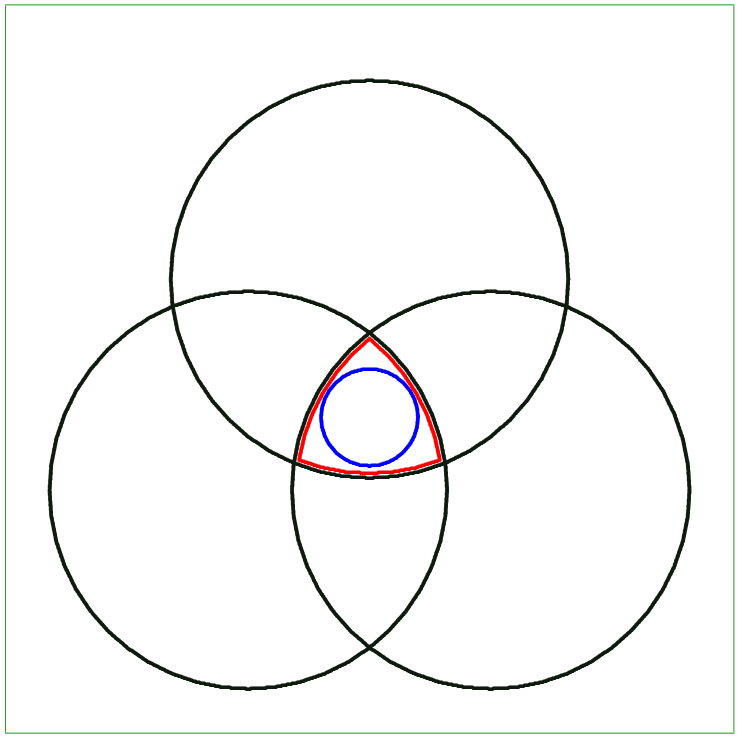
The three circles denote the sets U for which $f\ U \subseteq U$.
By hypothesis, $A \subseteq U$ for each U (1st, 2nd, 3rd . . .).
By definition, $lfp\ f$ is the intersection.
Clearly, $A \subseteq lfp\ f$.



Knaster-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 $A \subseteq lfp\ f$.

The three circles denote the sets U for which $f\ U \subseteq U$.
By hypothesis, $A \subseteq U$ for each U (1st, 2nd, 3rd . . .).
By definition, $lfp\ f$ is the intersection.
Clearly, $A \subseteq lfp\ f$.
Or as proof tree.



Knaster-Tarski's Fixpoint Theorem (3)

Claim 3: If f is monotone then $f(lfp\ f) \subseteq lfp\ f$.

Knaster-Tarski's Fixpoint Theorem (3)

Claim 3: If f is monotone then $f(lfp\ f) \subseteq lfp\ f$.

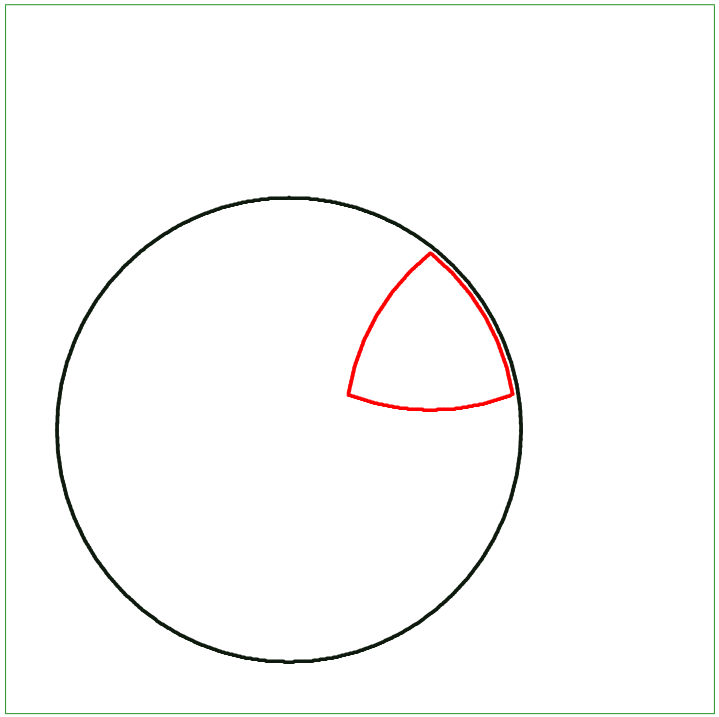
First show Claim 3*: $f\ U \subseteq U$ implies $f(lfp\ f) \subseteq U$.

Knaster-Tarski's Fixpoint Theorem (3)

Claim 3: If f is monotone then $f(lfp\ f) \subseteq lfp\ f$.

First show Claim 3*: $f\ U \subseteq U$ implies $f(lfp\ f) \subseteq U$.

Let the circle be such a U. By Claim 1, $lfp\ f \subseteq U$.



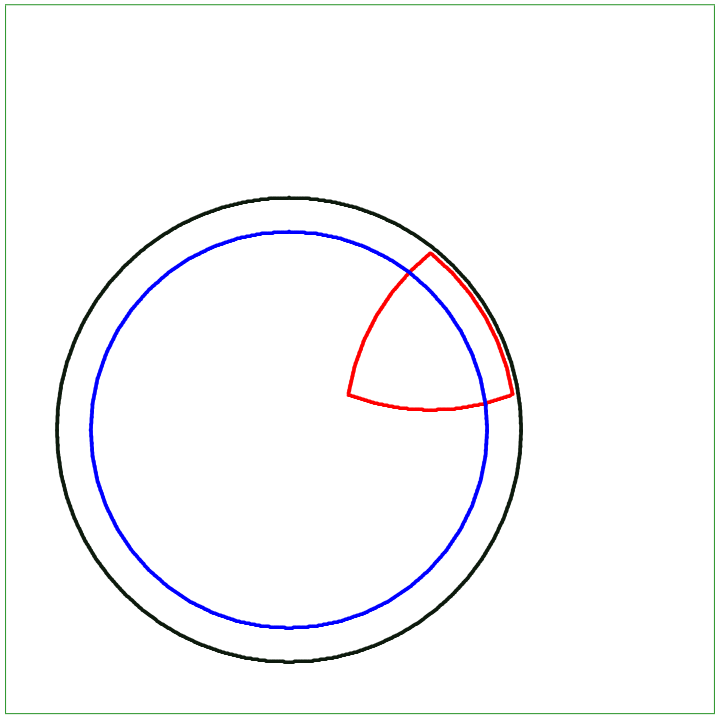
Or as proof tree.

Knaster-Tarski's Fixpoint Theorem (3)

Claim 3: If f is monotone then $f(lfp\ f) \subseteq lfp\ f$.

First show Claim 3*: $\underline{f\ U \subseteq U}$ implies $f(lfp\ f) \subseteq U$.

Let the circle be such a U. By Claim 1, $lfp\ f \subseteq U$.
$f\ U \subseteq U$ (<u>hypothesis</u>).



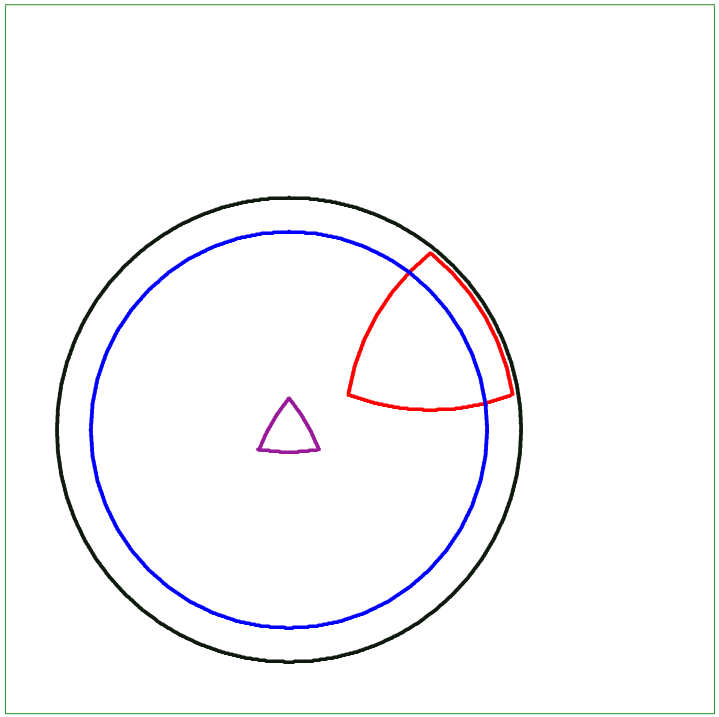
Or as proof tree.

Knaster-Tarski's Fixpoint Theorem (3)

Claim 3: If f is monotone then $f(lfp\ f) \subseteq lfp\ f$.

First show Claim 3*: $\underline{f\ U \subseteq U}$ implies $f(lfp\ f) \subseteq U$.

Let the circle be such a U. By Claim 1, $lfp\ f \subseteq U$.
$f\ U \subseteq U$ (hypothesis).
$f(lfp\ f) \subseteq f\ U$ (monotonicity).



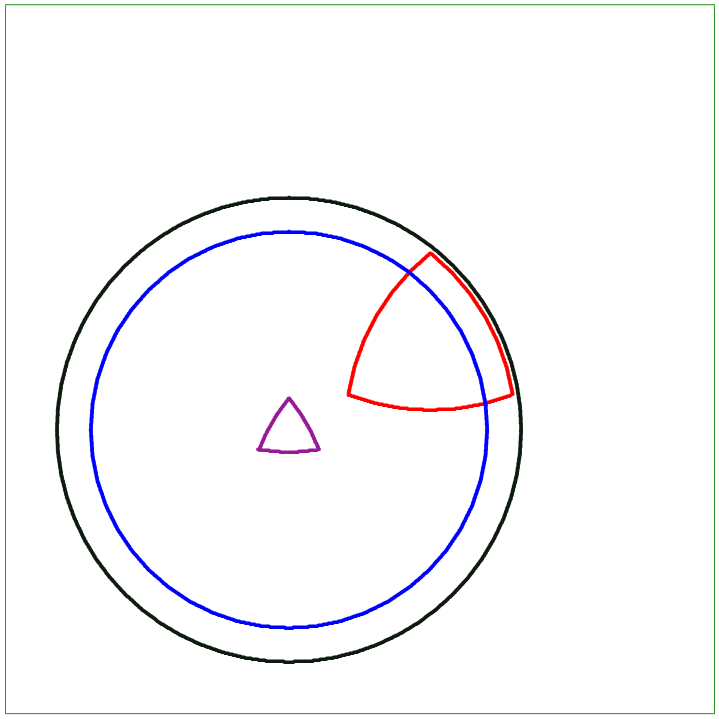
Or as proof tree.

Knaster-Tarski's Fixpoint Theorem (3)

Claim 3: If f is monotone then $f(lfp\ f) \subseteq lfp\ f$.

First show Claim 3*: $\underline{f\ U \subseteq U}$ implies $f(lfp\ f) \subseteq U$.

Let the circle be such a U. By Claim 1, $lfp\ f \subseteq U$.
$f\ U \subseteq U$ (hypothesis).
$f(lfp\ f) \subseteq f\ U$ (monotonicity).
$f(lfp\ f) \subseteq U$ (transitivity of $\subseteq$).
Claim 3* shown.



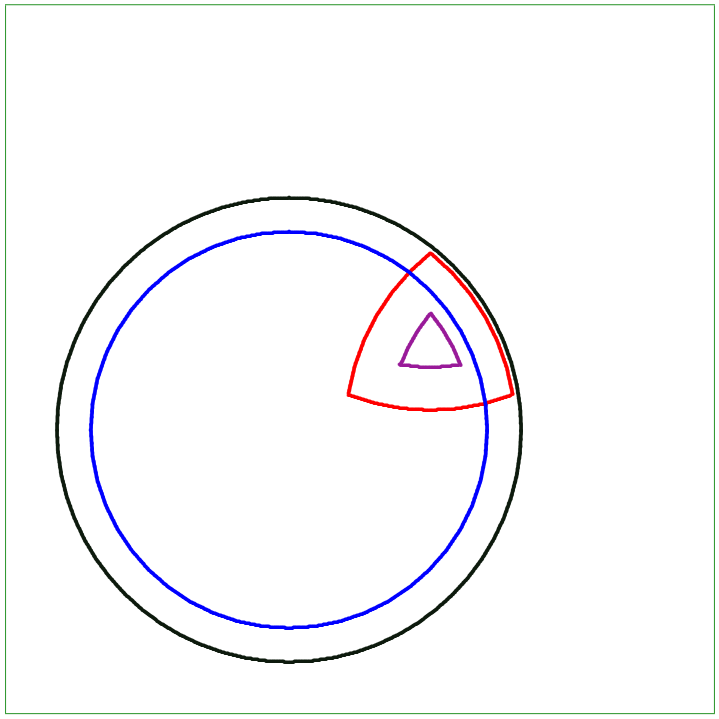
Or as proof tree.

Knaster-Tarski's Fixpoint Theorem (3)

Claim 3: If f is monotone then $f(lfp\ f) \subseteq lfp\ f$.

First show Claim 3*: $\underline{f\ U \subseteq U}$ implies $f(lfp\ f) \subseteq U$.

Let the circle be such a U. By Claim 1, $lfp\ f \subseteq U$.
$f\ U \subseteq U$ (hypothesis).
$f(lfp\ f) \subseteq f\ U$ (monotonicity).
$f(lfp\ f) \subseteq U$ (transitivity of $\subseteq$).
Claim 3* shown.
By Claim 2 (letting $A := f(lfp\ f)$), $f(lfp\ f) \subseteq lfp\ f$. Or as proof tree.



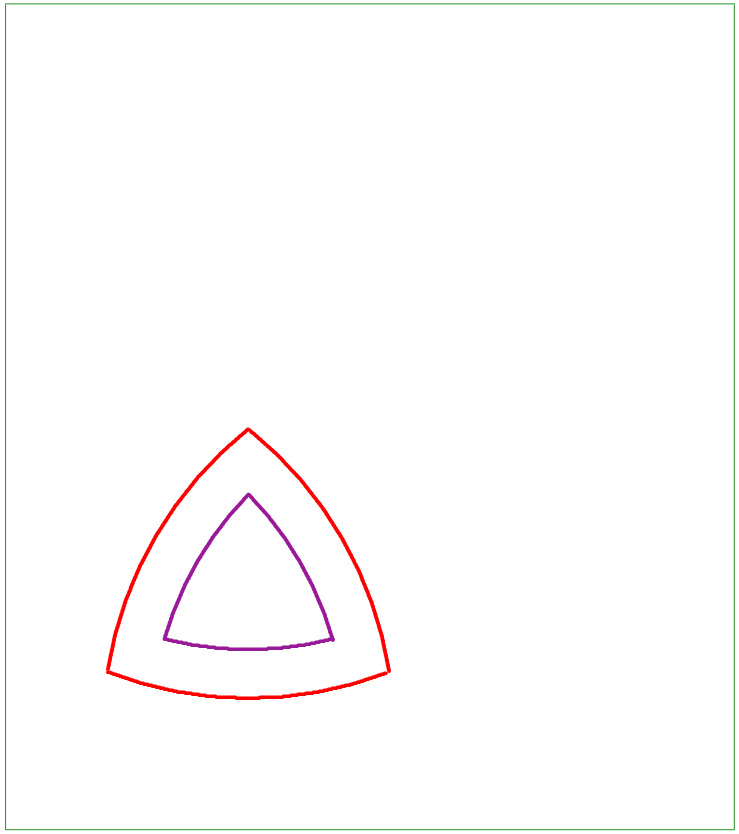
Knaster-Tarski's Fixpoint Theorem (4)

Claim 4: If f is monotone then $lfp\ f \subseteq f(lfp\ f)$.

Knaster-Tarski's Fixpoint Theorem (4)

Claim 4: If f is monotone then $lfp\ f \subseteq f(lfp\ f)$.

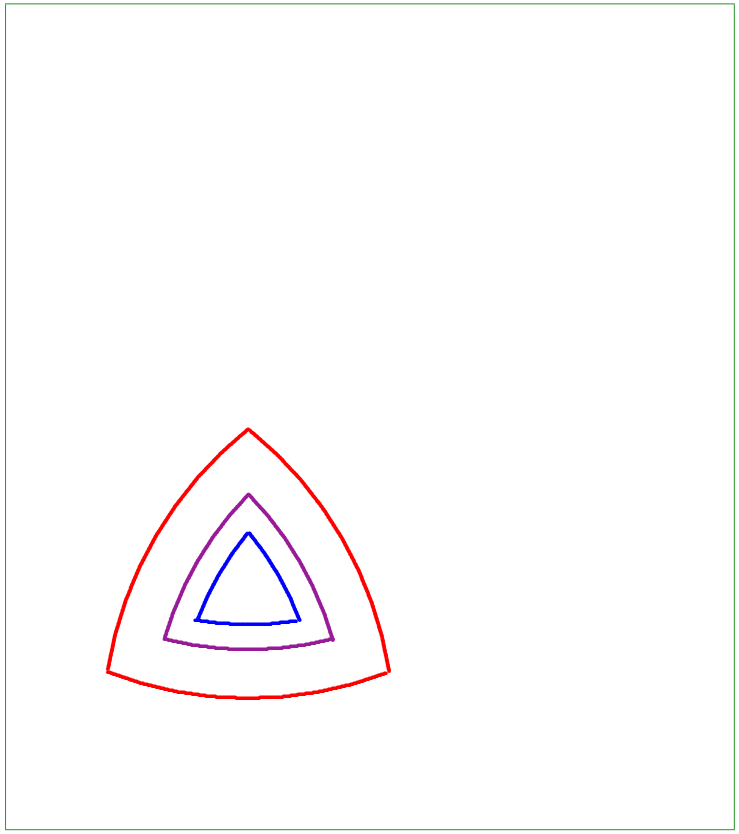
By Claim 3, $f(lfp\ f) \subseteq lfp\ f$.



Knaster-Tarski's Fixpoint Theorem (4)

Claim 4: If f is monotone then $lfp\ f \subseteq f(lfp\ f)$.

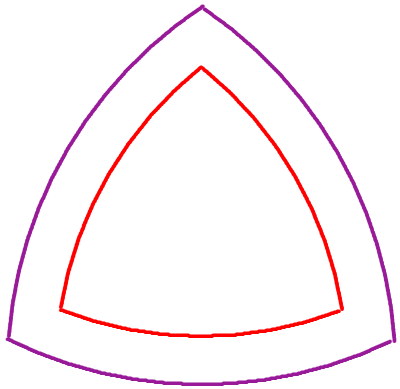
By Claim 3, $f(lfp\ f) \subseteq lfp\ f$.
By monotonicity, $f(f(lfp\ f)) \subseteq f(lfp\ f)$.



Knaster-Tarski's Fixpoint Theorem (4)

Claim 4: If f is monotone then $lfp\ f \subseteq f(lfp\ f)$.

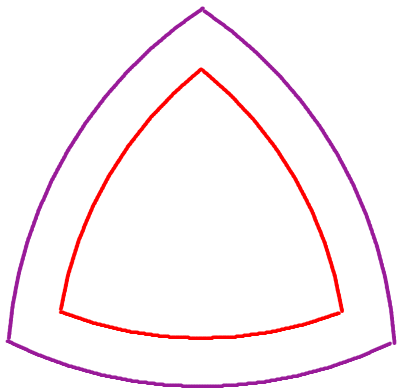
By Claim 3, $f(lfp\ f) \subseteq lfp\ f$.
By monotonicity, $f(f(lfp\ f)) \subseteq f(lfp\ f)$.
By Claim 1 (letting $A := f(lfp\ f)$), $lfp\ f \subseteq f(lfp\ f)$.



Knaster-Tarski's Fixpoint Theorem (4)

Claim 4: If f is monotone then $lfp\ f \subseteq f(lfp\ f)$.

By Claim 3, $f(lfp\ f) \subseteq lfp\ f$.
By monotonicity, $f(f(lfp\ f)) \subseteq f(lfp\ f)$.
By Claim 1 (letting $A := f(lfp\ f)$), $lfp\ f \subseteq f(lfp\ f)$.



Or as proof tree.

Knaster-Tarski's Fixpoint Theorem: QED

Claim 3 ($lfp\ f \subseteq f(lfp\ f)$) and Claim 4 ($f(lfp\ f) \subseteq lfp\ f$) together give the result:

If f is monotone, then $lfp\ f = f\ (lfp\ f)$.

So under appropriate conditions, lfp is a fixpoint combinator.

Or as proof tree.

We will reuse Claim 1 also for the proofs on induction.

Alternative: A Natural-Deduction Style Proof

The proof can also be presented in natural deduction style.

Knaster-Tarski's Fixpoint Theorem (1)

Claim 1 ("*lfp* lower bound"): If $f\ A \subseteq A$ then $lfp\ f \subseteq A$.

$$\dfrac{\dfrac{\dfrac{\dfrac{[f\ A \subseteq A]^1}{A \in \{u.fu \subseteq u\}}\ \text{CollectI}}{\bigcap\{u.fu \subseteq u\} \subseteq A}\ \text{Inter_lower}}{lfp\ f \subseteq A}\ \text{Def. } lfp}{f\ A \subseteq A \rightarrow lfp\ f \subseteq A}\ \rightarrow\text{-}I^1$$

Knaster-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 $A \subseteq \mathit{lfp}\ f$.

$$\dfrac{\dfrac{\dfrac{\dfrac{[\forall x. f x \subseteq x \rightarrow A \subseteq x]^1}{\forall x. x \in \{u. f u \subseteq u\} \rightarrow A \subseteq x}\ \textit{subst}, \text{CollectI}}{A \subseteq \bigcap\{u. f u \subseteq u\}}\ \text{Inter_greatest}}{A \subseteq \mathit{lfp} f}\ \text{Def. } \mathit{lfp}}{(\forall x. f x \subseteq x \rightarrow A \subseteq x) \rightarrow A \subseteq \mathit{lfp} f}\ \rightarrow\text{-}I^1$$

Knaster-Tarski's Fixpoint Theorem (3)

Claim 3: If f is monotone then $f(lfp\ f) \subseteq lfp\ f$.

$$
\begin{array}{rl}
 & \dfrac{[mono\ f]^1 \quad \dfrac{[f x \subseteq x]^2}{lfp\ f \subseteq x}}{f(lfp\ f) \subseteq f\ x} \qquad [f x \subseteq x]^2 \\
 & \hline
 & f(lfp\ f) \subseteq x \qquad \text{order_trans} \\
 & \hline
 & \forall x. f x \subseteq x \rightarrow f(lfp\ f) \subseteq x \qquad \forall\text{-}I,\ \rightarrow\text{-}I^2 \\
 & \hline
 & f(lfp\ f) \subseteq lfp\ f \qquad \text{Claim2},\ \rightarrow\text{-}E \\
 & \hline
 & mono\ f \rightarrow f(lfp\ f) \subseteq lfp\ f \qquad \rightarrow\text{-}I^1
\end{array}
$$

Knaster-Tarski's Fixpoint Theorem (4)

Claim 4: If f is monotone then $lfp\ f \subseteq f(lfp\ f)$.

$$\dfrac{\dfrac{\dfrac{[mono\ f]^1 \quad \dfrac{[mono\ f]^1}{f(lfp\ f) \subseteq lfp\ f}\ \text{Claim 3}, \rightarrow\text{-}E}{f(f(lfp\ f)) \subseteq f(lfp\ f)}\ \text{monoD}}{lfp\ f \subseteq f(lfp\ f)}\ \text{Claim1}, \rightarrow\text{-}E}{mono\ f \rightarrow lfp\ f \subseteq f(lfp\ f)}\ \rightarrow\text{-}I^1$$

Completing Proof Tree

$$\dfrac{\dfrac{\dfrac{[mono\ f]^1}{lfp\ f \subseteq f(lfp\ f)}\ \text{Claim 4} \qquad \dfrac{[mono\ f]^1}{f(lfp\ f) \subseteq lfp\ f}\ \text{Claim 3}}{lfp\ f = f(lfp\ f)}\ \text{equalityI}}{mono\ f \longrightarrow lfp\ f = f(lfp\ f)}\ \rightarrow\text{-}I^1$$

More Detailed Explanations

Algorithmic Version of lfp

The theorem

$$(\forall S.\ f(\bigcup S) = \bigcup(f\ `\ S)) \Longrightarrow \bigcup_{n \in N} (f^n\{\})) = lfp\ f$$

says that under a certain condition, $lfp\ f$ can be computed by applying f to the empty set over and over again:

- although the expression uses the union over all natural numbers, which is an infinite set, this can sometimes effectively be computed. Under certain conditions, there exists a k such that $f^k\ \{\} = f^{k+1}\{\}$.
- Even if $\bigcup n \in Nat.f^n\ \{\}$ cannot be effectively computed, it can still be approximated: for any k, we know that $\bigcup i \leq k.f^i\ \{\} \subseteq \bigcup n \in Nat.f^n\ \{\}$.

Monotone Functions

A function f is monotone w.r.t. a partial order $\leq$ if the following holds: $A \leq B$ implies $f(A) \leq f(B)$.

In particular, we consider the order given by the subset relation.

"The Set" α

α is not a set but a type (variable). But we can consider the set of all terms of that type (`UNIV` of type α).

The polymorphic constant `UNIV` was defined in `Set.thy`. `UNIV` of type τ `set` is the set containing all terms of type τ.

Three Circles?

In general, needless to say, there could be any number of such sets, but the picture is to be understood in the sense that the three circles are all the sets A with the property $f\ A \subseteq A$.

Different Phrasings

The theorem is phrased a bit differently in the "mathematical" version we give here and in the Isabelle version (see Lfp.thy). This is convenient for the graphical illustration of the proof.

The "mathematical phrasing" corresponding closely to the Isabelle version would be the following:

Theorem 6 (Induct (alternative)):

If

- $a \in \mathit{lfp}\ f$, and
- f is monotone, 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 7 (Induct (alternative)):

If

- $a \in \mathit{lfp}\ f$, and
- f is monotone, and
- $f(\mathit{lfp}\ f \cap \{x \mid P\,x\}) \subseteq \{x \mid P\,x\}$

then $P\ a$ holds.

Theorem 8 (Induct (alternative)):

If

- f is monotone, 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 $P\ x$.

Detail on Monotonicity

$\mathit{lfp}\ f \cap \{x \mid P\,x\} \subseteq \mathit{lfp}\ f$, so by monotonicity,
$f(\mathit{lfp}\ f \cap \{x \mid P\,x\}) \subseteq f(\mathit{lfp}\ f)$.

Use of Claim 1

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 (setting $A := \mathit{lfp}\ f \cap \{x \mid P\,x\}$), this implies $\mathit{lfp}(f) \subseteq \mathit{lfp}\ f \cap \{x \mid P\,x\}$.

Antisymmetry of $\subseteq$

We have $lfp\ f \cap \{x \mid P\,x\} \subseteq lfp(f)$ and $lfp(f) \subseteq lfp\ f \cap \{x \mid P\,x\}$, and so $lfp(f) = lfp\ f \cap \{x \mid P\,x\}$ by the antisymmetry of $\subseteq$.

Recursion in a Definitional Way

Recall why we were interested in fixpoints.

The problem with Y is that it leads to inconsistency (and of course, the definition of Y is not a constant definition/conservative extension.).

The definition of lfp is conservative.

And in appropriate situations, it can be used to define recursive functions.

Compared to Y, the type of lfp is restricted.

This restriction means that there is no obvious way to use lfp for defining recursive numeric functions such as fac.

Finite Sets in Isabelle

Above, we defined the set of finite subsets of a set A. Alternatively, one could define "the set of all finite sets whose elements have type τ". In this case, no fixed set A is involved, and it is closer to what actually happens in Isabelle.

In Finite_Set.thy a constant $Finites$ is defined. It has polymorphic type $\alpha\, set\, set$. We have $A \in Finites$ 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 of $Finites$ of type $\tau\, set\, set$ containing all finite sets of element type τ.

In Finite_Set.thy we find the inductive definition:

inductive " Finites"
  intros
    emptyI [simp, intro !]: "$\{\} \in$ Finites"

insertI [simp, intro !]: "A : Finites $\Longrightarrow$ *insert* a A $\in$ Finites"

The Isabelle mechanism of interpreting the keyword `inductive` translates this into the following definition: $Finites = lfp\ G$ where

$$G \equiv \lambda S.\ \{x \mid x = \{\} \vee (\exists A\ a.\ x = insert\ a\ A \wedge A \in S)\}$$

As a sanity-check, consider the type of this expression. The expression $insert\, a\, A$ forces A to be of type $\tau\ set$ for some τ and a to be of type τ. Next, $insert\, a\, A$ is of type $\tau\ set$, and hence x is also of type $\tau\ set$. Moreover, the expression $A \in S$ forces S to be of type $\tau\ set\ set$. The expression $\{x \mid x = \{\} \vee (\exists A\, a.\ x = insert\, a\, A \wedge A \in S)\}$ is of type $\tau\ set\ set$. Next, G is of type $\tau\ set\ set \rightarrow \tau\ set\ set$, and so finally, $Finites$ is of type $\tau\ set\ set$. But actually, since τ is arbitrary, we can replace it by a type variable α.

Note that there is a convenient syntactic translation

translations ”finite A” == ”A : Finites”

F is monotone

This proof is of course done in Isabelle.

Induction for Finite Sets

The theorem

$$[\![xa \in Fin\ A; P\ \{\}; \bigwedge ab.[\![a \in A; b \in Fin\ A; P\ b]\!] \Longrightarrow P\ (insert\ a\ b)]\!] \Longrightarrow P\ xa$$

is an instance of the general induction scheme. That is to say, if we take the general induction scheme `lfp_induct`

$$[\![a \in lfp\ f; mono\ f; \bigwedge x.x \in f(lfp\ f \cap \{x.P\ x\}) \Longrightarrow P\ x]\!] \Longrightarrow P\ a$$

and instantiate f to $\lambda X.\{\{\}\} \cup \bigcup x \in A.((\mathtt{insert}\ x)\ `\ X)$ then some massaging using the definitions will give us the first theorem.

Note here that monotonicity has disappeared from the assumptions. This is because the monotonicity of F is shown by Isabelle once and for all.

This is one aspect of what we mean by special proof support for inductive definitions.

The least fixpoint of the functional is $Fin\ A$ (the set of finite subsets of A) in this case.

Co-induction

Co-induction is a construction analogous to induction but using greatest fixpoints. This is a useful mechanism when defining infinite sequences, streams, and similar structures inductively.

Inductive Definitions

Jan-Georg Smaus and Burkhart Wolff

Induction Based on Lfp.thy

Theorem 9 (lfp induction):

If

- f is monotone, and
- $f(\mathit{lfp}\ f \cap \{x \mid P\,x\}) \subseteq \{x \mid P\,x\}$,

then $\mathit{lfp}\ f \subseteq \{x \mid P\,x\}$.

Induction Based on Lfp.thy

Theorem 9 (lfp induction):

If

- f is monotone, and
- $f(\mathit{lfp}\ f \cap \{x \mid P\,x\}) \subseteq \{x \mid P\,x\}$,

then $\mathit{lfp}\ f \subseteq \{x \mid P\,x\}$.

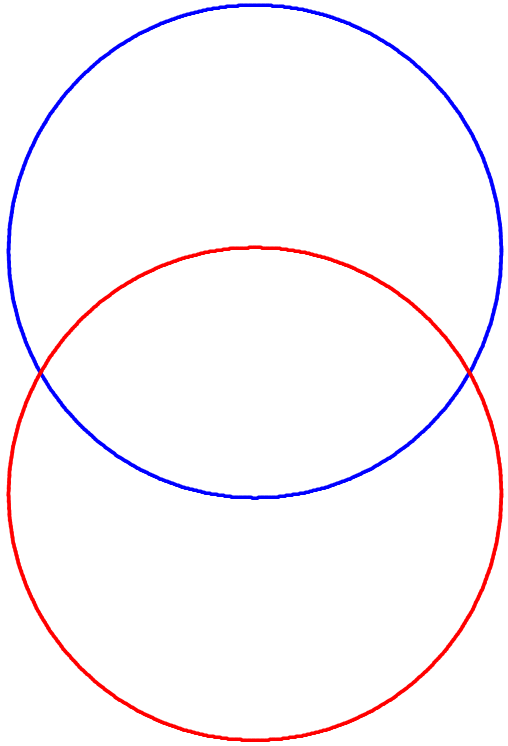
In Isabelle, it is called `lfp_induct`:

$$[\![a \in \mathit{lfp}\ f; \mathit{mono}\ f; \bigwedge x.x \in f(\mathit{lfp}\ f \cap \{x.P\,x\}) \Longrightarrow P\,x]\!] \Longrightarrow P\,a$$

We now show the theorem similarly as Tarski's Theorem.

Showing `lfp_induct`

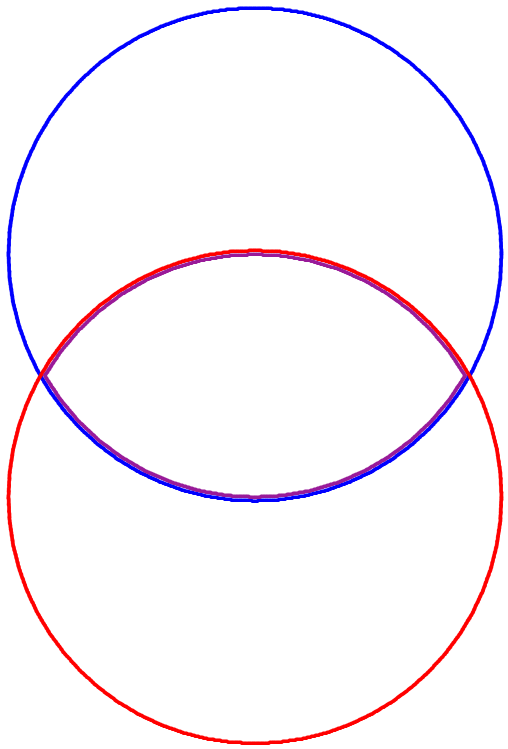
Circles denote $lfp\ f$ and $\{x \mid P\,x\}$.



Showing `lfp_induct`

Circles denote $lfp\ f$ and $\{x \mid P\,x\}$.

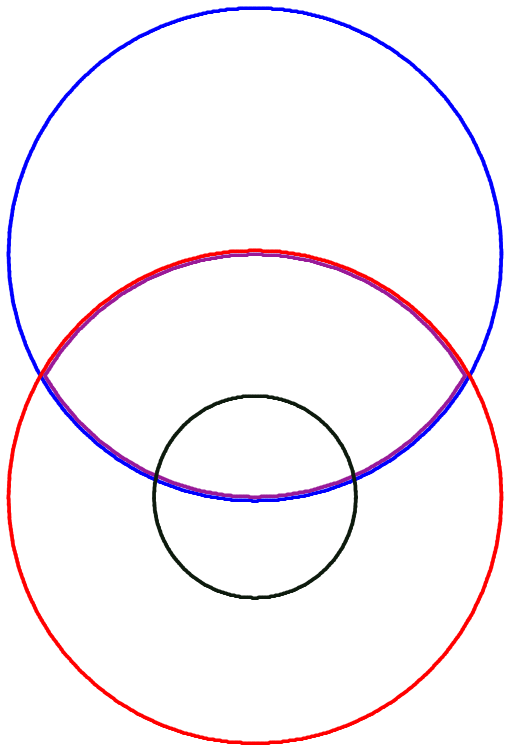
By monotonicity, $f(lfp\ f \cap \{x \mid P\,x\}) \subseteq f(lfp\ f)$. By Tarski, $lfp\ f = f(lfp\ f)$.



Showing `lfp_induct`

Circles denote $lfp\ f$ and $\{x \mid P\,x\}$.

By monotonicity, $f(lfp\ f \cap \{x \mid P\,x\}) \subseteq f(lfp\ f)$. By Tarski, $lfp\ f = f(lfp\ f)$. Hence $f(lfp\ f \cap \{x \mid P\,x\}) \subseteq lfp\ f$.

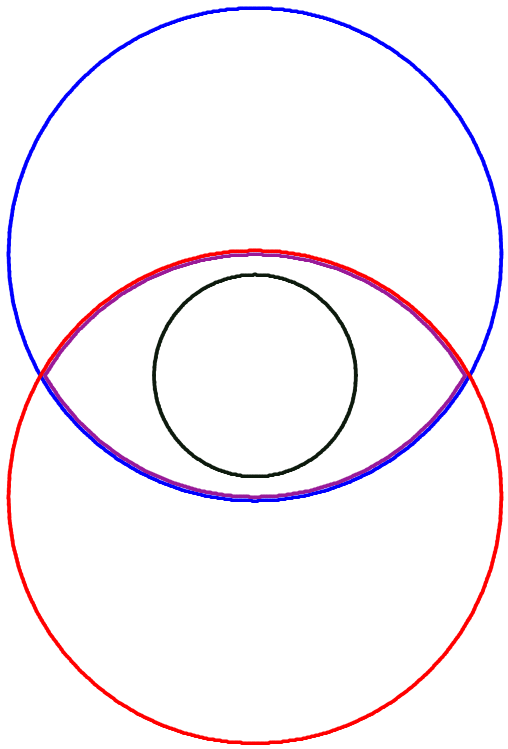


Showing `lfp_induct`

Circles denote $lfp\ f$ and $\{x \mid P\,x\}$.

By monotonicity, $f(lfp\ f \cap \{x \mid P\,x\}) \subseteq f(lfp\ f)$. By Tarski, $lfp\ f = f(lfp\ f)$. Hence $f(lfp\ f \cap \{x \mid P\,x\}) \subseteq lfp\ f$.

By hypothesis, $f(lfp\ f \cap \{x \mid P\,x\}) \subseteq \{x \mid P\,x\}$, and so we must adjust picture: $f(lfp\ f \cap \{x \mid P\,x\}) \subseteq lfp\ f \cap \{x \mid P\,x\}$.



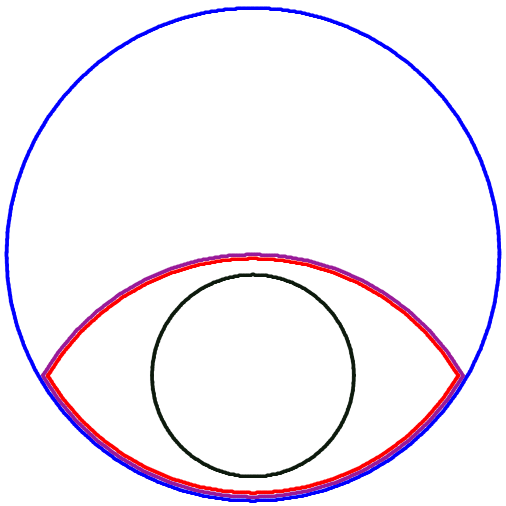
Showing `lfp_induct`

Circles denote $lfp\ f$ and $\{x \mid P\,x\}$.

By monotonicity, $f(lfp\ f \cap \{x \mid P\,x\}) \subseteq f(lfp\ f)$. By Tarski, $lfp\ f = f(lfp\ f)$. Hence $f(lfp\ f \cap \{x \mid P\,x\}) \subseteq lfp\ f$.

By hypothesis, $f(lfp\ f \cap \{x \mid P\,x\}) \subseteq \{x \mid P\,x\}$, and so we must adjust picture: $f(lfp\ f \cap \{x \mid P\,x\}) \subseteq lfp\ f \cap \{x \mid P\,x\}$.

By Claim 1, $lfp\ f \subseteq lfp\ f \cap \{x \mid P\,x\}$ and so $lfp\ f = lfp\ f \cap \{x \mid P\,x\}$.



Showing `lfp_induct`

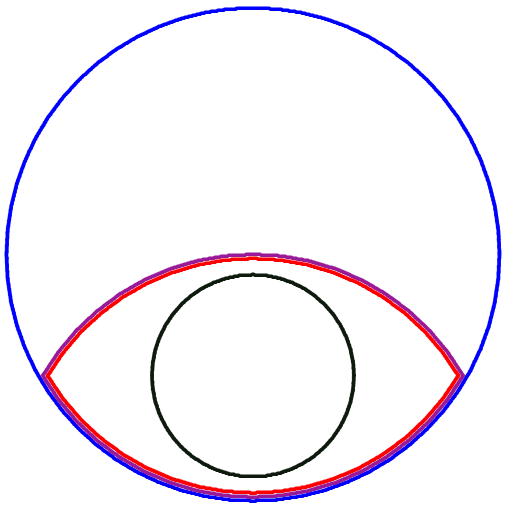
Circles denote $lfp\ f$ and $\{x \mid P\,x\}$.

By monotonicity, $f(lfp\ f \cap \{x \mid P\,x\}) \subseteq f(lfp\ f)$. By Tarski, $lfp\ f = f(lfp\ f)$. Hence $f(lfp\ f \cap \{x \mid P\,x\}) \subseteq lfp\ f$.

By hypothesis, $f(lfp\ f \cap \{x \mid P\,x\}) \subseteq \{x \mid P\,x\}$, and so we must adjust picture: $f(lfp\ f \cap \{x \mid P\,x\}) \subseteq lfp\ f \cap \{x \mid P\,x\}$.

By Claim 1, $lfp\ f \subseteq lfp\ f \cap \{x \mid P\,x\}$ and so $lfp\ f = lfp\ f \cap \{x \mid P\,x\}$.

Conclusion: $lfp\ f \subseteq \{x \mid P\,x\}$.



Approximating Fixpoints

Looking ahead: Suppose we have the set N of natural numbers (the type is formally introduced later). The theorem `approx`

$$(\forall S.\ f(\bigcup S) = \bigcup(f\ `\ S)) \Longrightarrow \bigcup_{n \in N}(f^n\{\}) = lfp\ f$$

shows a way of approximating lfp, which is important for algorithmic solutions (e.g. in program analysis).

You will show this as an exercise.

Where Are We Going? Induction and Recursion

Let's step back: What is an inductive definition of a set S?

Where Are We Going? Induction and Recursion

Let's step back: What is an inductive definition of a set S?

It has the form: S is the smallest set such that:

- $\emptyset \subseteq S$;
- if $S' \subseteq S$ then $F(S') \subseteq S$ (for some appropriate F).

Where Are We Going? Induction and Recursion

Let's step back: What is an inductive definition of a set S?

It has the form: S is the smallest set such that:

- $\emptyset \subseteq S$;
- if $S' \subseteq S$ then $F(S') \subseteq S$ (for some appropriate F).

At the same time, S is the smallest solution of the recursive equation $S = F(S)$.

Where Are We Going? Induction and Recursion

Let's step back: What is an inductive definition of a set S?

It has the form: S is the smallest set such that:

- $\emptyset \subseteq S$;
- if $S' \subseteq S$ then $F(S') \subseteq S$ (for some appropriate F).

At the same time, S is the smallest solution of the recursive equation $S = F(S)$.

Induction and recursion are two faces of the same coin.

Lfp.thy for Inductive Definitions

Least fixpoints are for building inductive definitions of sets in a definitional way: $S := lfp\ F$.

This is obviously well-defined, so why this fuss about monotonicity and Tarski?

Lfp.thy for Inductive Definitions

Least fixpoints are for building inductive definitions of sets in a definitional way: $S := lfp\ F$.

This is obviously well-defined, so why this fuss about monotonicity and Tarski?

Tarski allows us to exploit the equation $lfp\ f = f(lfp\ f)$ in proofs about S! That's what lfp is all about.

Example (Revisited)

The set of all finite subsets of a set A:

$$Fin\ A = lfp\ F$$

where $F = \lambda X.\{\{\}\} \cup \bigcup x \in A.((\texttt{insert}\ x)\ `\ X)$.

Example (Revisited)

The set of all finite subsets of a set A:

$$Fin\ A = lfp\ F$$

where $F = \lambda X.\{\{\}\} \cup \bigcup x \in A.((\texttt{insert}\ x)\ `\ X)$.

Thus we can do using lfp what we would have wanted to do using Y.

To show: F is monotone!

There will be an exercise on this.

Example: Transitive Closures

The transitive closure of a relation $R :: ('a \times' a)set$ can be defined as follows:

$$R^* = \mathit{lfp}\ F$$

where $F = \lambda X.Id \cup r\, O\, X)$.

Example: Transitive Closures

The transitive closure of a relation $R :: ('a \times' a)set$ can be defined as follows:

$$R^* = \mathit{lfp}\ F$$

where $F = \lambda X.Id \cup r\ O\ X)$.

To show: F is monotone!

There is also an exercise on this.

The Package for Inductive Sets

Since inductive definitions are so ubiquitious, Isabelle provides a special compiler (a "package") with an own front-end syntax.

The Package for Inductive Sets

Since inductive definitions are so ubiquitious, Isabelle provides a special compiler (a "package") with an own front-end syntax.

It generates conservative definitions and derives introduction and induction rules from them.

The Package for Inductive Sets

Example:

The Package for Inductive Sets

Example:

```
consts Fin :: 'a set => 'a set set
inductive "Fin(A)"
  intrs
    emptyI "{} : Fin(A)"
    insertI "[| a: A; b: Fin(A) |] ⟹
insert a b : Fin(A)"
```

Technical Support for Inductive Definitions

Support important in practice since many constructions are based on inductively defined sets (datatypes, . . .) Support provided for:

- Automatic proof of monotonicity
- Automatic proof of induction rule, for example:

$$[\![xa \in Fin\ A; P\ \{\}; \bigwedge ab.[\![a \in A; b \in Fin\ A; P\ b]\!] \Longrightarrow P\ (insert\ a\ b)]\!] \Longrightarrow P\ xa$$

This works also for mutually recursive definitions, co-inductive definitions, . . .

Summary on Least Fixpoints

- We are interested in recursion because we need recursively defined sets.
- It turns out that inductively defined sets are solutions to recursive equations.
- We cannot have general fixpoint operator Y, but we can have a conservatively defined least fixpoints operator.
- There is an induction scheme (lfp induction) for proving theorems about an inductively defined sets.

Higher-Order Logic: Well-Founded Recursion

Burkhart Wolff

The Roadmap

We are still looking at how the different parts of mathematics are encoded in the Isabelle/HOL library.

- Orders
- Sets
- Functions
- (Least) fixpoints and induction
- (Well-founded) recursion
- Arithmetic
- Datatypes

The Roadmap

We are still looking at how the different parts of mathematics are encoded in the Isabelle/HOL library.

- Orders
- Sets
- Functions
- (Least) fixpoints and induction
- (Well-founded) recursion
- Arithmetic
- Datatypes

Motivation

Motivation(1)

After least fixpoints, well-founded recursion is our second concept of recursion

Motivation(1)

After least fixpoints, well-founded recursion is our second concept of recursion represented by another fixpoint combinator.

Motivation(1)

After least fixpoints, well-founded recursion is our second concept of recursion represented by another fixpoint combinator.

Idea: Modeling "terminating" recursive functions, i.e. recursive definitions that use "smaller" arguments for the recursive call.

Claim: An axiom like:

$$fac = (\lambda n.\ \mathbf{if}\ n = 0\ \mathbf{then}\ 1\ \mathbf{else}\ n * fac(n-1))$$

is no problem since "it terminates" !

Motivation(2)

However: Logic talks about validity, not execution !

Moreover: is this true?

Motivation(2)

However: Logic talks about validity, not execution !

Moreover: is this true? What does this mean precisely ?

Motivation(2)

However: Logic talks about validity, not execution !

Moreover: is this true? What does this mean precisely ?

1. Consider: fac :: int $\longrightarrow$ int !

Motivation(2)

However: Logic talks about validity, not execution !

Moreover: is this true? What does this mean precisely ?

1. Consider: fac :: int $\longrightarrow$ int !
2. Consider:

$$fac = (\lambda n.\ \mathbf{if}\ n = 0\ \mathbf{then}\ 1\ \mathbf{else}\ n * fac(n+1))$$

Motivation(2)

However: Logic talks about validity, not execution !

Moreover: is this true? What does this mean precisely ?

1. Consider: fac :: int $\longrightarrow$ int !
2. Consider:

$$fac = (\lambda n.\ \mathbf{if}\ n = 0\ \mathbf{then}\ 1\ \mathbf{else}\ n * fac(n+1))$$

Motivation(3)

1) shows that arguments must be ordered wrt. to a well-founded ("terminating") ordering,

Motivation(3)

1) shows that arguments must be ordered wrt. to a well-founded ("terminating") ordering,
2) shows that the context of the recursive call ("the function body") must be coherent, i.e. it must supply only arguments to the recursive call which are lesser w.r.t. this ordering.

Motivation(3)

1) shows that arguments must be ordered wrt. to a well-founded ("terminating") ordering,
2) shows that the context of the recursive call ("the function body") must be coherent, i.e. it must supply only arguments to the recursive call which are lesser w.r.t. this ordering.

How can this be modeled?

Motivation(4)

One aspect of the problem: In HOL we can represent the "context of a recursive call".

Motivation(4)

One aspect of the problem: In HOL we can represent the "context of a recursive call". Reconsider:

$$fac = (\lambda n.\ \mathbf{if}\ n = 0\ \mathbf{then}\ 1\ \mathbf{else}\ n * fac(n-1))$$

Motivation(4)

One aspect of the problem: In HOL we can represent the "context of a recursive call". Reconsider:

$$fac = (\lambda n.\ \mathbf{if}\ n = 0\ \mathbf{then}\ 1\ \mathbf{else}\ n * fac(n-1))$$

Abstracting the recursive call yields:

$$Fac = (\lambda f.\ \lambda n.\ \mathbf{if}\ n = 0\ \mathbf{then}\ 1\ \mathbf{else}\ n * f(n-1))$$

We say: Fac is the body of fac.

Motivation(4)

One aspect of the problem: In HOL we can represent the "context of a recursive call". Reconsider:

$$fac = (\lambda n.\ \mathbf{if}\ n = 0\ \mathbf{then}\ 1\ \mathbf{else}\ n * fac(n-1))$$

Abstracting the recursive call yields:

$$Fac = (\lambda f.\ \lambda n.\ \mathbf{if}\ n = 0\ \mathbf{then}\ 1\ \mathbf{else}\ n * f(n-1))$$

We say: Fac is the body of fac.

Recall that a general fixpoint combinator can define fac by its body by $Y\ Fac$ and thus solve $fac = Fac\ fac$.

In the sequel, we will define and explore the

- concept of well-founded ordering
- concept of coherence of a body

Prerequisite: Relations

We need some standard operations on binary relations (sets of pairs), such as converse, composition, image of a set and a relation, the identity relation, . . .

These are provided by Relation.thy.

Relation.thy (Fragment)

constdefs

```
converse  ::  ('a×'b)set ⇒ ('b×'a) set    ("(_^−1)" ..)
r^−1      ≡{(y, x). (x, y) ∈ r}
rel_comp  ::  [('b×'c)set, ('a×'b)set] ⇒ ('a×'c)set
                                          ("(_O_)" ..)
r O s ≡{(x,z). ∃ y. (x, y)∈s ∧ (y, z)∈r}
Image ::  [('a×'b)set, 'a set] ⇒ 'b set   ("(_“_)" ..)
r “ s ≡{y. ∃ x∈s. (x,y)∈r}
Id    ::  ('a×'a) set
Id    ≡{p. ∃ x. p = (x,x)}
```

As can be expected, these notions are similar to Fun.thy.

Prerequisite: Closures

We need the transitive, as well as the reflexive transitive closure of a relation.

These are provided by Transitive_Closure.thy.

How would you define those inductively

Transitive_Closure.thy (Fragment)

consts

rtrancl :: ('a × 'a) set ⇒ ('a × 'a) set
("(_^*)" ..)

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 ("(_^+)" ..)

inductive "r^+"

intros

r_into_trancl [...]:

(a, b) ∈ r ⟹ (a, b) ∈ r^+

trancl_into_trancl [...]:

(a, b) ∈ r^+ ⟹ (b, c) ∈ r ⟹ (a,c) ∈ r^+

Well-Founded Orderings

Defined in Wellfounded_Recursion.thy.

Wellfounded_Recursion = Transitive_Closure +

constdefs

$$\begin{aligned} &\mathsf{wf} \quad :: \ ('a \times 'a)\ \mathsf{set} \Rightarrow \mathsf{bool} \\ &\mathsf{wf}(r) \ \equiv \ (\forall\ P.\ (\forall\ x.\ (\forall\ y.\ (y,x) \in r \longrightarrow P(y)) \\ &\qquad\qquad \longrightarrow P(x)) \longrightarrow (\forall\ x.\ P(x))) \end{aligned}$$

In other words . . .

Well-Founded Orderings

Defined in Wellfounded_Recursion.thy.

Wellfounded_Recursion = Transitive_Closure +

constdefs

$$\begin{aligned} &\mathsf{wf} \quad :: \ (\text{'a} \times \text{'a})\ \mathsf{set} \Rightarrow \mathsf{bool} \\ &\mathsf{wf}(\mathsf{r}) \ \equiv \ (\forall\ \mathsf{P}.\ (\forall\ \mathsf{x}.\ (\forall\ \mathsf{y}.\ (\mathsf{y},\mathsf{x}) \in \mathsf{r} \longrightarrow \mathsf{P}(\mathsf{y})) \\ &\qquad\qquad \longrightarrow \mathsf{P}(\mathsf{x})) \longrightarrow (\forall\ \mathsf{x}.\ \mathsf{P}(\mathsf{x}))) \end{aligned}$$

In other words . . . A relation r is well-founded iff well-founded (Noetherian) induction based on r is a valid proof scheme.

Well-Founded Orderings

Defined in Wellfounded_Recursion.thy.

Wellfounded_Recursion = Transitive_Closure +

constdefs

$$\begin{aligned} &\mathsf{wf} \quad :: \ (\text{'a} \times \text{'a})\ \mathsf{set} \Rightarrow \mathsf{bool} \\ &\mathsf{wf}(r) \ \equiv \ (\forall\ P.\ (\forall\ x.\ (\forall\ y.\ (y,x) \in r \longrightarrow P(y)) \\ &\qquad\qquad \longrightarrow P(x)) \longrightarrow (\forall\ x.\ P(x))) \end{aligned}$$

In other words . . . A relation r is well-founded iff well-founded (Noetherian) induction based on r is a valid proof scheme. This is conservative, fine. But does it meet our intuition of "termination"?

Gaining Intuition of Well-Foundedness

A first reality-check: Is $\emptyset$ well-founded?

Gaining Intuition of Well-Foundedness

A first reality-check: Is $\emptyset$ well-founded?

The definition of wf is:

$$wf(r) \equiv \forall P.(\forall x.(\forall y.(y,x) \in r \rightarrow P(y)) \rightarrow P(x)) \rightarrow (\forall x.P(x))$$

Gaining Intuition of Well-Foundedness

A first reality-check: Is $\emptyset$ well-founded?

The definition of wf is:

Let's instantiate r to $\emptyset$.

$$wf(\emptyset) \equiv \forall P.(\forall x.(\forall y.(y,x) \in \emptyset \rightarrow P(y)) \rightarrow P(x)) \rightarrow (\forall x.P(x))$$

Gaining Intuition of Well-Foundedness

A first reality-check: Is $\emptyset$ well-founded?

The definition of wf is:

Let's instantiate r to $\emptyset$.

$$wf(\emptyset) \equiv \forall P.(\forall x.(\forall y.False \qquad \rightarrow P(y)) \rightarrow P(x)) \rightarrow (\forall x.P(x))$$

Gaining Intuition of Well-Foundedness

A first reality-check: Is $\emptyset$ well-founded?

The definition of wf is:

Let's instantiate r to $\emptyset$.

$$wf(\emptyset) \equiv \forall P.(\forall x.(\forall y.True \qquad\qquad) \rightarrow P(x)) \rightarrow (\forall x.P(x))$$

Gaining Intuition of Well-Foundedness

A first reality-check: Is $\emptyset$ well-founded?

The definition of wf is:

Let's instantiate r to $\emptyset$.

$$wf\,(\emptyset) \equiv \forall P.\,True$$

Gaining Intuition of Well-Foundedness

A first reality-check: Is $\emptyset$ well-founded?

The definition of wf is:

Let's instantiate r to $\emptyset$.

$$wf\,(\emptyset) \equiv \qquad True$$

So the empty set is well-founded.

Gaining Intuition of Well-Foundedness

Intuition of wf: All descending chains are finite.

But: concept of "finite chain" is difficult to express; we therefore look for for alternatives.

- Not symmetric:

Gaining Intuition of Well-Foundedness

Intuition of wf: All descending chains are finite.

But: concept of "finite chain" is difficult to express; we therefore look for for alternatives.

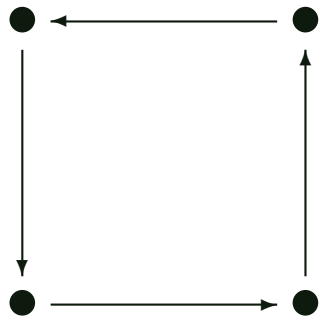
- Not symmetric: $(x, y) \in r \rightarrow (y, x) \notin r$?

Gaining Intuition of Well-Foundedness

Intuition of wf: All descending chains are finite.

But: concept of "finite chain" is difficult to express; we therefore look for for alternatives.

- Not symmetric: $(x, y) \in r \rightarrow (y, x) \notin r$?



Gaining Intuition of Well-Foundedness

Intuition of wf: All descending chains are finite.

But: concept of "finite chain" is difficult to express; we therefore look for for alternatives.

- Not symmetric: $(x, y) \in r \rightarrow (y, x) \notin r$?
- No cycles:

Gaining Intuition of Well-Foundedness

Intuition of wf: All descending chains are finite.

But: concept of "finite chain" is difficult to express; we therefore look for for alternatives.

- Not symmetric: $(x, y) \in r \rightarrow (y, x) \notin r$?
- No cycles: $(x, x) \notin r^+$?

Gaining Intuition of Well-Foundedness

Intuition of wf: All descending chains are finite.

But: concept of "finite chain" is difficult to express; we therefore look for for alternatives.

- Not symmetric: $(x, y) \in r \rightarrow (y, x) \notin r$?
- No cycles: $(x, x) \notin r^+$?

Gaining Intuition of Well-Foundedness

Intuition of wf: All descending chains are finite.

But: concept of "finite chain" is difficult to express; we therefore look for for alternatives.

- Not symmetric: $(x, y) \in r \rightarrow (y, x) \notin r$?
- No cycles: $(x, x) \notin r^+$?
- r has minimal element:

Gaining Intuition of Well-Foundedness

Intuition of wf: All descending chains are finite.

But: concept of "finite chain" is difficult to express; we therefore look for for alternatives.

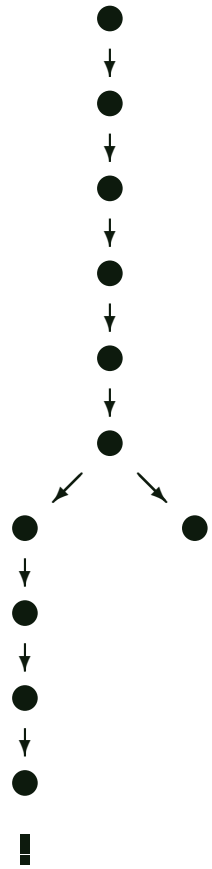
- Not symmetric: $(x, y) \in r \rightarrow (y, x) \notin r$?
- No cycles: $(x, x) \notin r^+$?
- r has minimal element: $\exists x. \forall y. (y, x) \notin r$?
  Note: Trivial for $r = \emptyset$.

Gaining Intuition of Well-Foundedness

Intuition of wf: All descending chains are finite.

But: concept of "finite chain" is difficult to express; we therefore look for for alternatives.

- Not symmetric: $(x, y) \in r \rightarrow (y, x) \notin r$?
- No cycles: $(x, x) \notin r^+$?
- r has minimal element: $\exists x.\forall y.(y, x) \notin r$?
  Note: Trivial for $r = \emptyset$.



Gaining Intuition of Well-Foundedness

Intuition of wf: All descending chains are finite.

But: concept of "finite chain" is difficult to express; we therefore look for for alternatives.

- Not symmetric: $(x, y) \in r \rightarrow (y, x) \notin r$?
- No cycles: $(x, x) \notin r^+$?
- r has minimal element: $\exists x.\forall y.(y, x) \notin r$?
  Note: Trivial for $r = \emptyset$.
- Any subrelation must have minimal element:

Gaining Intuition of Well-Foundedness

Intuition of wf: All descending chains are finite.

But: concept of "finite chain" is difficult to express; we therefore look for for alternatives.

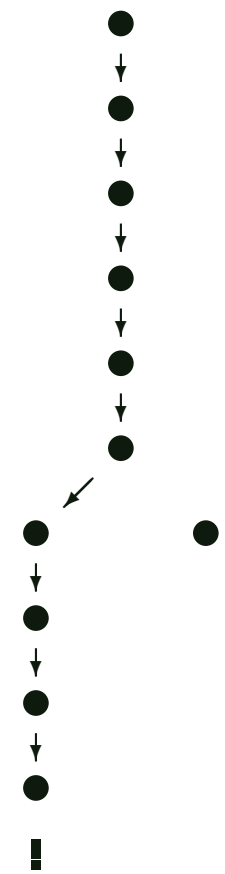
- Not symmetric: $(x, y) \in r \rightarrow (y, x) \notin r$?
- No cycles: $(x, x) \notin r^+$?
- r has minimal element: $\exists x.\forall y.(y, x) \notin r$?
  Note: Trivial for $r = \emptyset$.
- Any subrelation must have minimal element:
  $\forall p.p \subseteq r \rightarrow \exists x.\forall y.(y, x) \notin p$?

Gaining Intuition of Well-Foundedness

Intuition of wf: All descending chains are finite.

But: concept of "finite chain" is difficult to express; we therefore look for for alternatives.

- Not symmetric: $(x, y) \in r \longrightarrow (y, x) \notin r$?
- No cycles: $(x, x) \notin r^+$?
- r has minimal element: $\exists x.\forall y.(y, x) \notin r$?
  Note: Trivial for $r = \emptyset$.
- Any subrelation must have minimal element: $\forall p.p \subseteq r \longrightarrow \exists x.\forall y.(y, x) \notin p$? "Minimal element" badly formalized (already in previous point).



The Characterisation

All these attempts are just necessary but not sufficient conditions for well-foundedness.

Here is a characterization:

$$wf\, r = \forall r'.\, r' \neq \{\} \wedge r' \subseteq r \longrightarrow (\exists x \in Domain\, r'. \forall y. (y, x) \notin r')$$

Here is an alternative characterization:

$$wf\, r = (\forall Q x.\ x \in Q \longrightarrow (\exists x \in Q.\ \forall y. (y, x) \in r \longrightarrow y \notin Q))$$

Let's see some theorems to confirm our intuition, including the statements just shown.

A Theorem for Induction

By massage of the definition of well-foundedness

$$\forall P.(\forall x.(\forall y.(y,x) \in r \longrightarrow P\,y) \longrightarrow P\,x) \longrightarrow (\forall x.P\,x)$$

one obtains the theorem wf_induct

$$[\![wf\ r; \bigwedge x.\forall y.(y,x) \in r \longrightarrow P\,y \Longrightarrow P\,x]\!] \Longrightarrow P\,a.$$

This is a form suitable for doing induction proofs in Isabelle.

Induction Theorem as Proof Rule

The Isabelle theorem wf_induct

$$[\![wf\ r; \bigwedge x.\forall y.(y,x) \in r \longrightarrow P\,y \Longrightarrow P\,x]\!] \Longrightarrow P\,a.$$

as proof rule:

$$\frac{wf\ r \qquad \begin{array}{c}[\forall y.(y,x) \in r \longrightarrow P\,y] \\ \vdots \\ P\,x\end{array}}{P\,a}\ \texttt{wf_induct}$$

A Theorem on Antisymmetry

wf_not_sym: \<lbrakk wf r; (a, x) **in** r⟧ ⟹ (x, a) ∈ r

Proof sketch:

$$\frac{wf\ r \qquad \begin{array}{c}[\forall y.(y,x) \in r \rightarrow (\forall z.(y,z) \in r \rightarrow (z,y) \notin r)] \\ \vdots \\ \forall z.(x,z) \in r \rightarrow (z,x) \notin r\end{array}}{\forall z.(a,z) \in r \rightarrow (z,a) \notin r}\ \texttt{wf_induct}$$

Rest routine though not so trivial (needs classical reasoning).
A variation will be done as exercise.

Theorems on Absence of Cycles

wf_not_refl : wf r $\Longrightarrow$ (a, a) $\notin$ r
wf_trancl : wf r $\Longrightarrow$ wf (r^+)
wf_acyclic : wf r $\Longrightarrow$ acyclic r

(where $acyclic\, r \equiv \forall x.(x, x) \notin r^+$)

Proof sketch:

wf_not_refl : Corollary of wf_not_sym.
wf_trancl : Uses induction.
wf_acyclic : Apply wf_not_refl and wf_trancl.

Ergo: Definition of wf meets our intuition of "no cycles".

Another Theorem ("Exists Minimal Element")

wf_minimal: wf r $\Longrightarrow \exists$ x. $\forall$ y. (y,x) $\notin$ r^+

Proof sketch, abbreviating $\phi \equiv (\exists x.\forall y.(y,x) \notin r^+)$:

$$\frac{\mathtt{wf}(r)}{\phi}\ \mathtt{wf_minimal}$$

This is what we must construct.

Another Theorem ("Exists Minimal Element")

wf_minimal: wf r $\Longrightarrow \exists$ x. $\forall$ y. (y,x) $\notin$ r^+

Proof sketch, abbreviating $\phi \equiv (\exists x.\forall y.(y,x) \notin r^+)$:

$$\forall w.(w,v) \in r^+ \rightarrow \phi$$

$$\frac{\begin{array}{c}\mathtt{wf}(r)\\ \mathtt{wf}(r^+)\end{array} \qquad \phi}{\phi}\ \mathtt{wf_induct}$$

Note "special case": w and v do not occur in ϕ!

Another Theorem ("Exists Minimal Element")

wf_minimal: wf r $\Longrightarrow \exists$ x. $\forall$ y. (y,x) $\notin$ r^+

Proof sketch, abbreviating $\phi \equiv (\exists x.\forall y.(y,x) \notin r^+)$:

$$\forall w.(w,v) \in r^+ \to \phi$$

$$\dfrac{\dfrac{\mathtt{wf}(r)}{\mathtt{wf}(r^+)}\bullet \qquad \phi}{\phi}\ \mathtt{wf_induct}$$

This is wf_trancl .

Another Theorem ("Exists Minimal Element")

wf_minimal: wf r $\Longrightarrow \exists$ x. $\forall$ y. (y,x) $\notin$ r^+

Proof sketch, abbreviating $\phi \equiv (\exists x.\forall y.(y,x) \notin r^+)$:

$$\neg\phi \qquad \neg\phi \qquad \begin{array}{c}\forall w.(w,v)\\ \in r^+ \to \phi\end{array}$$

$$\dfrac{\dfrac{\mathtt{wf}(r)}{\mathtt{wf}(r^+)} \bullet \quad \dfrac{\phi \vee \neg\phi \qquad \phi \qquad\qquad \phi}{\phi}\,\textit{disjE}}{\phi}\,\mathtt{wf_induct}$$

We now try a proof by case distinction on ϕ.

Another Theorem ("Exists Minimal Element")

wf_minimal: wf r $\Longrightarrow \exists$ x. $\forall$ y. (y,x) $\notin$ r^+

Proof sketch, abbreviating $\phi \equiv (\exists x.\forall y.(y,x) \notin r^+)$:

$$\neg\phi \qquad \neg\phi \qquad \begin{array}{c}\forall w.(w,v)\\ \in r^+ \to \phi\end{array}$$

$$\dfrac{\dfrac{\mathtt{wf}(r)}{\mathtt{wf}(r^+)}\bullet \qquad \dfrac{\dfrac{}{\phi \vee \neg\phi}\bullet \qquad \phi \qquad \phi}{\phi}\,\textit{disjE}}{\phi}\,\mathtt{wf_induct}$$

Classical reasoning.

Another Theorem ("Exists Minimal Element")

wf_minimal: wf r $\Longrightarrow \exists$ x. $\forall$ y. (y,x) $\notin$ r^+

Proof sketch, abbreviating $\phi \equiv (\exists x.\forall y.(y,x) \notin r^+)$:

$$\dfrac{\dfrac{\dfrac{\mathtt{wf}(r)}{\mathtt{wf}(r^+)}\bullet \qquad \dfrac{\dfrac{}{\phi \vee \neg\phi}\bullet \qquad \phi \qquad \dfrac{\dfrac{\neg\phi}{\forall x.\exists y.(y,x) \in r^+}\cdots \qquad \neg\phi \qquad \begin{array}{c}\forall w.(w,v)\\ \in r^+ \to \phi\end{array}}{\phi}}{\phi}\,\textit{disjE}}{\phi}\,\mathtt{wf_induct}$$

Using some elementary equivalences.

Another Theorem ("Exists Minimal Element")

wf_minimal: wf r $\Longrightarrow \exists$ x. $\forall$ y. (y,x) $\notin$ r^+

Proof sketch, abbreviating $\phi \equiv (\exists x.\forall y.(y,x) \notin r^+)$:

$$\dfrac{\neg\phi}{\forall x.\exists y.(y,x) \in r^+} \cdots \quad \dfrac{\neg\phi \qquad \begin{array}{c}\forall w.(w,v)\\ \in r^+ \to \phi\end{array}}{\exists w.(w,v) \notin r^+} \bullet$$

$$\dfrac{\dfrac{\mathtt{wf}(r)}{\mathtt{wf}(r^+)}\bullet \qquad \dfrac{\dfrac{}{\phi \vee \neg\phi}\bullet \qquad \phi \qquad\qquad \phi}{\phi}\,\textit{disjE}}{\phi}\,\mathtt{wf_induct}$$

This step works for any ϕ. Think semantically or check!

Another Theorem ("Exists Minimal Element")

wf_minimal: wf r $\Longrightarrow \exists$ x. $\forall$ y. (y,x) $\notin$ r^+

Proof sketch, abbreviating $\phi \equiv (\exists x.\forall y.(y,x) \notin r^+)$:

$$\dfrac{\dfrac{\dfrac{\mathtt{wf}(r)}{\mathtt{wf}(r^+)}\bullet \qquad \dfrac{\dfrac{}{\phi \vee \neg\phi}\bullet \qquad \phi \qquad \dfrac{\dfrac{\dfrac{\neg\phi}{\forall x.\exists y.(y,x) \in r^+} \cdots \qquad \dfrac{\neg\phi \qquad \dfrac{\forall w.(w,v) \in r^+ \to \phi}{\exists w.(w,v) \notin r^+}\bullet}{False}}{\phi}\,\mathit{FalseE}}{\phi}\,\mathit{disjE}}{\phi}\,\mathtt{wf_induct}}{\phi}$$

It is routine to derive $False$.

Another Theorem ("Exists Minimal Element")

wf_minimal: wf r $\Longrightarrow \exists$ x. $\forall$ y. (y,x) $\notin$ r^+

Proof sketch, abbreviating $\phi \equiv (\exists x.\forall y.(y,x) \notin r^+)$:

$$\dfrac{\dfrac{\dfrac{\mathtt{wf}(r)}{\mathtt{wf}(r^+)}\bullet \qquad \dfrac{\dfrac{}{\phi \vee \neg\phi}\bullet \qquad [\phi]^2 \qquad \dfrac{\dfrac{\dfrac{[\neg\phi]^2}{\forall x.\exists y.(y,x) \in r^+} \cdots \qquad \dfrac{[\neg\phi]^2 \qquad \begin{array}{c}\forall w.(w,v)\\ \in r^+ \to \phi\end{array}}{\exists w.(w,v) \notin r^+}\bullet}{False}}{\phi}\,\textit{FalseE}}{\phi}\,\textit{disjE}^2}{\phi}\,\mathtt{wf_induct}$$

This completes the proof by case distinction . . .

Another Theorem ("Exists Minimal Element")

wf_minimal: wf r $\Longrightarrow \exists$ x. $\forall$ y. (y,x) $\notin$ r^+

Proof sketch, abbreviating $\phi \equiv (\exists x.\forall y.(y,x) \notin r^+)$:

$$\dfrac{\dfrac{\mathtt{wf}(r)}{\mathtt{wf}(r^+)}\bullet \qquad \dfrac{\dfrac{}{\phi \vee \neg\phi}\bullet \qquad [\phi]^2 \qquad \dfrac{\dfrac{\dfrac{[\neg\phi]^2}{\forall x.\exists y.(y,x) \in r^+} \cdots \qquad \dfrac{[\neg\phi]^2 \quad [\forall w.(w,v) \in r^+ \to \phi]^1}{\exists w.(w,v) \notin r^+}\bullet}{False}}{\phi}\,\mathit{FalseE}}{\phi}\,\mathit{disjE}^2 \cdots}{\phi}\,\mathtt{wf_induct}^1$$

. . . and the proof by induction.

A Characterization of wf

The theorem wf_eq_minimal is characterization of well-foundedness.:

$$wf\ r = (\forall Q x.x \in Q \longrightarrow (\exists z \in Q.\forall y.(y,z) \in r \longrightarrow y \notin Q))$$

Proof uses iffI $=$, use `wf_def`, rest routine.

Ergo: Definition of `wf` meets textbook definitions "every non-empty set Q has a minimal element in r" (more or less standard textbook).

A Theorem on Subsets

`wf_subset` $[\![wf\ r; p \subseteq r]\!] \Longrightarrow wf\ p$

Proof sketch:

wf_subset: simplification tactic using wf_eq_minimal.

A Theorem on Subrelations

$$\text{wf } r \Longrightarrow \forall\, p.\ p \subseteq r \longrightarrow \exists\, x.\ \forall\, y.\ (y,x) \notin p\hat{}+$$

A Theorem on Subrelations

wf r $\Longrightarrow$ $\forall$ p. p $\subseteq$ r $\longrightarrow$ $\exists$ x. $\forall$ y. (y,x) $\notin$ p^+

Proof sketch: Combine wf_minimal and wf_subset.

This implies $wf\ r \Longrightarrow \forall p.p \subseteq r \rightarrow \exists x.\forall y.(x,y) \notin p$.

A Theorem on Subrelations

wf r $\Longrightarrow \forall$ p. p $\subseteq$ r $\longrightarrow \exists$ x. $\forall$ y. (y,x) $\notin$ p^+

Proof sketch: Combine wf_minimal and wf_subset.
This implies $wf\ r \Longrightarrow \forall p.p \subseteq r \rightarrow \exists x.\forall y.(x,y) \notin p$.

Ergo: `wf` fulfills the conditions of second attempt of characterizing well-foundedness using minimal elements.

A Theorem on Subrelations

wf r $\Longrightarrow$ ∀ p. p ⊆ r $\longrightarrow$ ∃ x. ∀ y. (y,x) ∉ p^+

Proof sketch: Combine wf_minimal and wf_subset.
This implies $wf\ r \Longrightarrow \forall p.p \subseteq r \rightarrow \exists x.\forall y.(x,y) \notin p$.

Ergo: `wf` fulfills the conditions of second attempt of characterizing well-foundedness using minimal elements.
Note this is not a characterization: The subrelation must be non-empty, and minimum must be in the domain of p in order to rule out an isolated element, unrelated to the subrelation. (see characterizations)

Defining Recursive Functions

Coherent Function Bodies

A function body H is coherent w.r.t. $<$ if all recursive calls are supplied with arguments "smaller" than the original argument.

This means that $H\,f\,a$ and $H\,f'\,a$ are equal provided that that $f\,x = f'\,x$ for all $x < a$.

This allows us to use an approximation f' instead of a "perfect" f when recursively defining a function.

Using Approximating f's

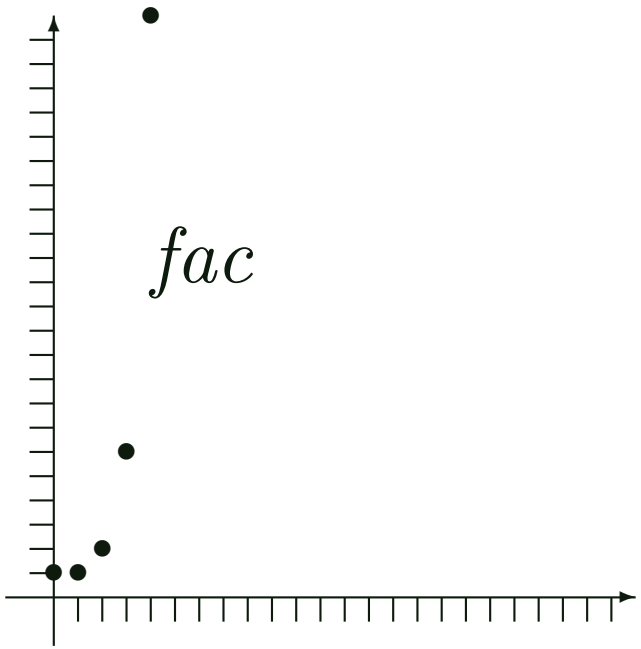
Let $f|_{<a}$ be a function that is like f on all values $< a$, and arbitrary elsewhere. $f|_{<a}$ is an approximation, a "bad" f.

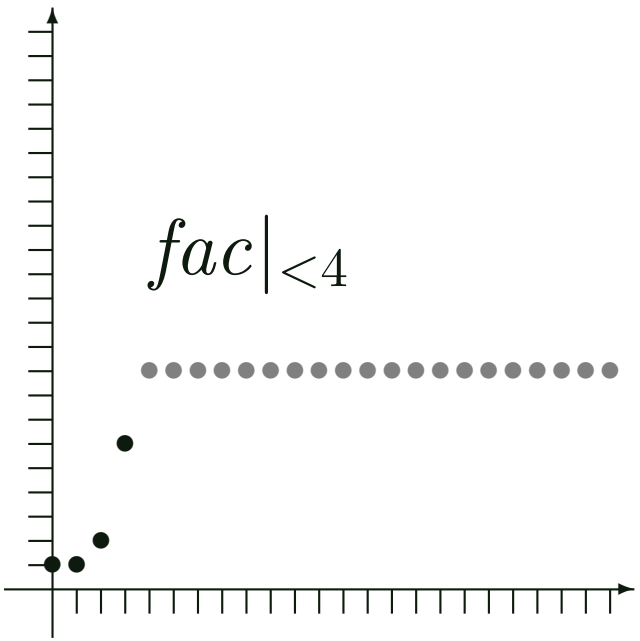
Now we can define coherence of H by:

$$H\, f\, a = H\, (f|_{<a})\, a. \tag{5}$$

Approximating f's: Example

Consider fac. On the right-hand side, we show one possibility for $fac|_{<4}$):





cut (in Wellfounded_Recursion.thy)

Technically, the function $f|_{<x}$ is defined as follows:

constdefs

cut :: ('a ⇒ 'b) ⇒ ('a × 'a)set ⇒ 'a ⇒ 'a ⇒ 'b
cut f r x ≡ λy. if (y,x)∈r then f y else arbitrary

cut (in Wellfounded_Recursion.thy)

Technically, the function $f|_{<x}$ is defined as follows:

constdefs

cut :: ('a ⇒ 'b) ⇒ ('a × 'a)set ⇒ 'a ⇒ 'a ⇒ 'b
cut f r x ≡ λy. if (y,x)∈r then f y else arbitrary

The unspecified constant `arbitrary` is declared in HOL.thy.

cut (in Wellfounded_Recursion.thy)

Technically, the function $f|_{<x}$ is defined as follows:

constdefs

cut :: ('a ⇒ 'b) ⇒ ('a × 'a)set ⇒ 'a ⇒ 'a ⇒ 'b
cut f r x ≡ λy. if (y,x)∈r then f y else arbitrary

The unspecified constant `arbitrary` is declared in HOL.thy.

The function cut f r x is therefore unspecified for arguments y where (y,x)∉ r, but for each such argument, (cut f r x) y must be the same in any particular model.

Theorems Involving cut

Properties of cut:

cuts_eq	$(\mathsf{cut}\ \mathsf{f}\ \mathsf{r}\ \mathsf{x} = \mathsf{cut}\ \mathsf{g}\ \mathsf{x}) =$
	$(\forall\ \mathsf{y}.\ (\mathsf{y},\mathsf{x}) \in \mathsf{r} \longrightarrow\ \mathsf{f}\ \mathsf{y} = \mathsf{g}\ \mathsf{y})$
cut_apply	$(\mathsf{x},\mathsf{a}) \in \mathsf{r} \Longrightarrow\ \mathsf{cut}\ \mathsf{f}\ \mathsf{r}\ \mathsf{a}\ \mathsf{x} = \mathsf{f}\ \mathsf{x}$

Theorems Involving cut

Properties of cut:

cuts_eq	(cut f r x = cut g x) =
	(∀ y. (y,x)∈r ⟶ f y = g y)
cut_apply	(x,a)∈r ⟹ cut f r a x = f x

Or, using the previous textbook notation:

`cuts_eq`	$(f\|_{<x} = g\|_{<x}) = (\forall y.y < x \longrightarrow f\,y = g\,y)$
`cut_apply`	$x < a \Longrightarrow f\|_{<a}\,x = f\,x$

wfrec_rel (in Wellfounded_Recursion.thy)

construction: "approximate" f by a relation wfrec_rel R F.

wfrec_rel :: ('a × 'a) set ⇒
(('a ⇒ 'b) ⇒ 'a ⇒ 'b) ⇒ ('a × 'b) set

inductive " wfrec_rel R F"
intrs
wfrecI ∀ z. (z, x) ∈ R ⟶ (z, g z) ∈ wfrec_rel R F
⟹ (x, F g x) ∈ wfrec_rel R F

More on wfrec_rel

Assume the ordering on natural numbers pred_nat and assume wf pred_nat.

More on wfrec_rel

Assume the ordering on natural numbers pred_nat and assume wf pred_nat.

Question: Which elements do we have in wfrec_rel pred_nat Fac ?

More on wfrec_rel

Assume the ordering on natural numbers pred_nat and assume wf pred_nat.

Question: Which elements do we have in wfrec_rel pred_nat Fac ?

$(0,\ \text{Fac g } 0) \in \text{wfrec_rel pred_nat Fac}$
$(1,\ \text{Fac (Fac g) } 1) \in \text{wfrec_rel pred_nat Fac}$
$(2,\ \text{Fac (Fac (Fac g)) } 2) \in \text{wfrec_rel pred_nat Fac}$
. . .

wfrec (in Wellfounded_Recursion.thy)

Now we turn the relation wfrec_rel into a function:

$$\text{wfrec} :: ('a \times 'a)\ \text{set} \Rightarrow (('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'b$$

$$\text{wfrec}\ R\ F \equiv \lambda x.\ \text{THE}\ y.\ (x,\ y) \in \text{wfrec_rel}\ R\ (\lambda f\ x.\ F(\text{cut}\ f\ R\ x)x)$$

Note that the type of wfrec R is again an instance of the type of the Y-combinator (similar lfp).

THE x. P x picks the unique a such that P a holds, if it exists. Otherwise (see HOL.thy) it is arbitrary.

The Fixpoint Theorem

Theorem: wfrec satisfies the fixpoint property:

$$\text{wfrec}: \ \text{wf r} \Longrightarrow \text{wfrec r H a} = \text{H (cut wfrec r H r a) a}$$

The Fixpoint Theorem

Theorem: wfrec satisfies the fixpoint property:

$$\text{wfrec}: \ \text{wf r} \Longrightarrow \text{wfrec r H a} = \text{H (cut wfrec r H r a) a}$$

Note that wfrec is used here both as a name of a constant (defined above) and a theorem.

The Fixpoint Theorem

Theorem: wfrec satisfies the fixpoint property:

$$\text{wfrec}: \ \text{wf r} \Longrightarrow \text{wfrec r H a} = \text{H (cut wfrec r H r a) a}$$

Note that wfrec is used here both as a name of a constant (defined above) and a theorem. So if R is well-founded and the body H is coherent, we have

$$\text{wfrec r H a} = \text{H (wfrec r H) a}$$

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.

wfrec is applied to a well-founded order and a body to define a function.

Example for *wfrec*: Natural Numbers

The constant wfrec provides the mechanism/support for defining recursive functions. We illustrate this using nat, the type of natural numbers.

wfrec is applied to a well-founded order and a body to define a function.

First, define predecessor relation:

constdefs

$$\text{pred_nat} :: (\text{nat} \times \text{nat})\ \text{set}$$
$$\text{pred_nat} \equiv \{(\text{m},\text{n}).\ \text{n} = \text{Suc m}\}$$

How would you define addition or subtraction?

Defining Division and Modulus

```
div :: ['a::div, 'a] ⇒ 'a                    (infixl 70)
m div n ≡wfrec (pred_nat^+)
                (λf 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^+)
                (λf j. if j<n ∨n=0 then j
                     else f (j−n)) m
```

Here, `div` is a syntactic class for which division is defined.

We assume a definition for −(subtract).

The functions are recursive in one argument (just like `add`).

Theorems of the Example

wf_pred_nat: wf pred_nat

$$\mathsf{m\ mod\ n} = \mathsf{if\ m} < \mathsf{n\ then\ m\ else\ (m} - \mathsf{n)\ mod\ n}$$
$$\mathsf{m\ div\ n} = \mathsf{if\ m} < \mathsf{n\ then\ 0\ else\ Suc((m} - \mathsf{n)\ div\ n)}$$

Theorems of the Example

wf_pred_nat: wf pred_nat

m mod n = if m < n then m else (m − n) mod n
m div n = if m < n then 0 else Suc((m − n) div n)

This is very similar to functional programming code and hence lends itself to real computations (rewriting), as opposed to only doing proofs.

Package for Primitive Recursion

For primitive recursion, finding a well-founded ordering is simple enough for automation!

Package for Primitive Recursion

For primitive recursion, finding a well-founded ordering is simple enough for automation!

Examples (use `nat` and `case`-syntax): . . .

Recursion and Arithmetic

Isabelle provides a syntactic front-end for defining an important subclass of well-founded recursions, namely primitive recursive functions:

```
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)
```

Recursion and Arithmetic

recdef statement is more general and requires a mesure-function (involving a proof of well-foundedness potentially requiring user interaction).

Example:

```
consts posDivAlg :: "int*int => int*int"
recdef posDivAlg "inv_image less_than
                    (λ(a,b). nat(a − b + 1))"
 "posDivAlg (a,b) = (if (a<b | b≤0) then (0,a)
                      else adjust b (posDivAlg(a, 2*b)))"
```

Conclusion

- We can model recursively defined functions conservatively!
- Together with the theory of least fixpoints, we can avoid a general fixpoint combinator Y.
- There is a further powerful induction principle wf_induct.
- The methodological overhead can be faced by powerful mechanical support.

More Detailed Explanations

Bad Formalization of "Minimal Element"

In this attempt, we formalized the "minimal element in p" as an x such that there is no y with $(x, y) \in p$. But this is a bad formalization since an isolated element, i.e., one that is completely unrelated to p, or even to r, would meet the definition.

In fact, this problem was already present for the previous attempt where we just required $\exists x.\forall y.(y, x) \notin r$ (i.e., r has a minimal element).

No Infinite Descending Chains

The final condition

$$(\forall Q x.x \in Q \longrightarrow (\exists z \in Q.\forall y.(y,z) \in r \longrightarrow y \notin Q))$$

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 "offical" definition is somewhat indirect since it defines well-foundedness by an induction principle. As we have seen, both repesentations are equivalent.

induct_wf

As far as the induction principle is concerned, `induct_wf` states the same as the very definition of `wf`. All that happens is that some explicit universal object-level quantifiers are removed and the according variables are (implicitly) universally quantified on the meta-level, and some shifting from object-level implications to meta-level implications using `mp`. This is why we dare say "logical massage". See `Wellfounded_Recursion.ML`.

Elementary Equivalences

For example $\neg\forall x.\phi = \exists x.\neg\phi$ or $\neg\neg\phi = \phi$, which hold because our reasoning is classical.

$\neg\exists w.(w,v) \in r^+$ in Detail

In the proof of $\exists x.\forall y.(y,x) \notin r^+$ we had the sub-proof

$$\frac{\neg\phi \quad \forall w.(w,v) \in r^+ \to \phi}{\neg\exists w.(w,v) \in r^+}$$

This sub-proof does not actually depend on ϕ, it would hold no matter what ϕ is (unlike the entire proof)

In detail, the sub-proof looks as follows:

$$\frac{\forall w.(w,v) \in r^+ \to \phi}{(w,v) \in r^+ \to \phi}\ \textit{spec}$$

In detail, the sub-proof looks as follows:

$$(w,v) \in r^+ \qquad \frac{\forall w.(w,v) \in r^+ \to \phi}{(w,v) \in r^+ \to \phi}\ \textit{spec}$$

In detail, the sub-proof looks as follows:

$$\dfrac{(w,v) \in r^+ \qquad \dfrac{\forall w.(w,v) \in r^+ \to \phi}{(w,v) \in r^+ \to \phi}\,\textit{spec}}{\phi}\,\textit{mp}$$

In detail, the sub-proof looks as follows:

$$\exists w.(w,v) \in r^+ \qquad \dfrac{(w,v) \in r^+ \qquad \dfrac{\forall w.(w,v) \in r^+ \to \phi}{(w,v) \in r^+ \to \phi}\,\textit{spec}}{\phi}\,\textit{mp}$$

In detail, the sub-proof looks as follows:

$$\dfrac{\exists w.(w,v) \in r^+ \qquad \dfrac{[(w,v) \in r^+]^2 \qquad \dfrac{\forall w.(w,v) \in r^+ \to \phi}{(w,v) \in r^+ \to \phi}\,\textit{spec}}{\phi}\,\textit{mp}}{\phi}\,\textit{exE}^2$$

In detail, the sub-proof looks as follows:

$$\neg\phi \qquad \dfrac{\exists w.(w,v) \in r^+ \qquad \dfrac{[(w,v) \in r^+]^2 \qquad \dfrac{\forall w.(w,v) \in r^+ \to \phi}{(w,v) \in r^+ \to \phi}\,\textit{spec}}{\phi}\,\textit{mp}}{\phi}\,\textit{exE}^2$$

In detail, the sub-proof looks as follows:

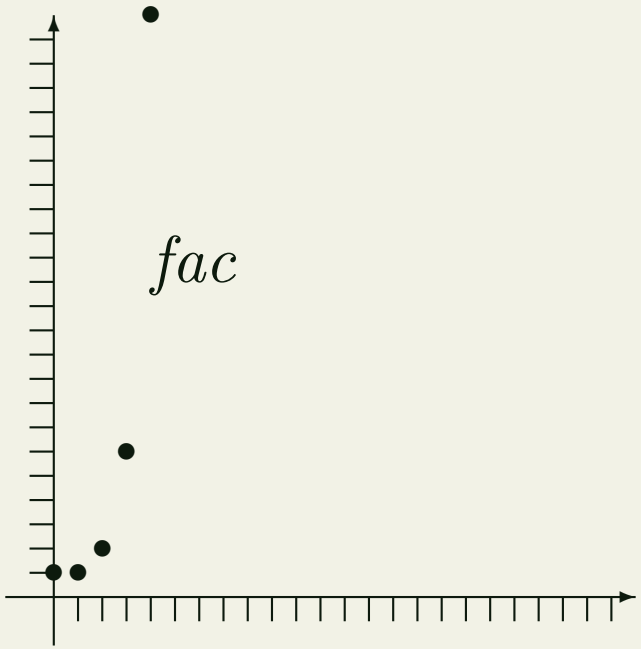
$$\dfrac{\neg\phi \qquad \dfrac{\exists w.(w,v) \in r^+ \qquad \dfrac{[(w,v) \in r^+]^2 \qquad \dfrac{\forall w.(w,v) \in r^+ \to \phi}{(w,v) \in r^+ \to \phi}\,\textit{spec}}{\phi}\,\textit{mp}}{\phi}\,\textit{exE}^2}{\mathit{False}}\,\textit{notE}$$

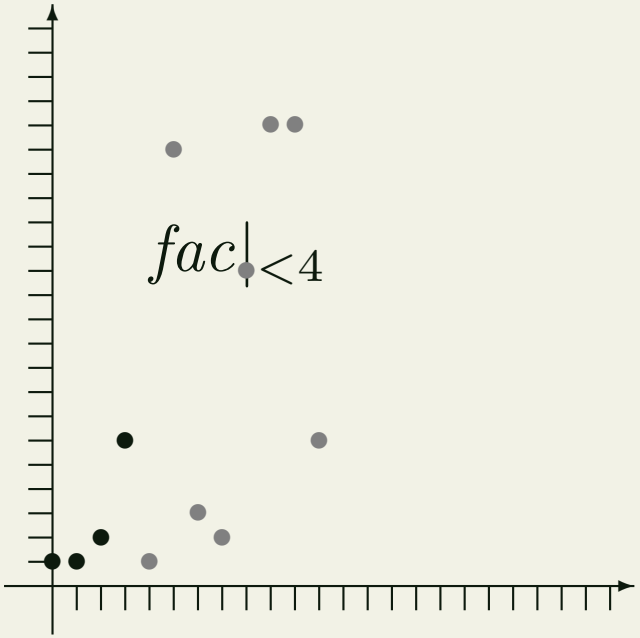
In detail, the sub-proof looks as follows:

$$\dfrac{\dfrac{\neg\phi \qquad \dfrac{[\exists w.(w,v) \in r^+]^1 \qquad \dfrac{[(w,v) \in r^+]^2 \qquad \dfrac{\forall w.(w,v) \in r^+ \to \phi}{(w,v) \in r^+ \to \phi}\,\textit{spec}}{\phi}\,\textit{mp}}{\phi}\,\textit{exE}^2}{\mathit{False}}\,\textit{notE}}{\neg\exists w.(w,v) \in r^+}\,\textit{notI}^1$$

Appoximating Functions by cut?

For the construction we have in mind, it would be fine that $f|_{<a}$ be a function that is like f on all values $< a$, and arbitrary elsewhere. E.g., $fac|_{<4}$ could be





However, such a $fac|_{<4}$ could not be in a model for HOL. Since arbitrary is an uninterpreted constant declared in HOL.thy, it turns out

that in any model and for each type, there must be one specific element in the semantic domain for it. 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.

Relation is a Function

When we say that a binary relation $r : \tau \times \sigma$ is in fact a function, we mean that for $t : \tau$, there is exactly one $s : \sigma$ such that $(t, s) \in r$.

Define Addition and Subtraction

```
add :: [nat, nat] ⇒ nat                    ( infixl 70)
m add n ≡wfrec (pred_nat^+)
                  (λ f j. if j=0 then n
                          else Suc(f(pred j))) m
```

Here we suppose that we have a predecessor function `pred` (which can be defined using the Hilbert-operator).

Note that `add` is a function of type $nat \to nat \to nat$ (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^+)
      (λ f j. if j=0 then n else Suc(f(pred j)))
```

By the definitions (of *wfrec* most importantly), this expression is a function of type $nat \to nat$, 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^+)
               (λf 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 that for `add`.

Similar considerations apply for other binary functions defined by recursion in one argument.

Primitive Recursion

A function is primitive recursive if the recursion is based on the immediate predecessor w.r.t. the well-founded order used (e.g., the predecessor on the natural numbers, as opposed to any arbitrary smaller numbers).

This is not the same concept as used in the context of computation theory, where primitive recursive is in contrast to μ-recursive [LP81].

Automated Support of Recursive Functions

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. The ordering is limited, but the proof will be automatic.

recdef statement is more general and requires a mesure-function (involving a proof of well-foundedness potentially requiring user interaction). Example:

```
consts posDivAlg :: "int*int => int*int"
recdef posDivAlg "inv_image less_than (λ(a,b). nat(a − b + 1))"
 "posDivAlg (a,b) = (if (a<b | b≤ 0) then (0,a)
                     else adjust b (posDivAlg(a, 2*b)))"
```

Higher-Order Logic: Arithmetic

Burkhart Wolff

The Roadmap

We are still looking at how the different parts of mathematics are encoded in the Isabelle/HOL library.

- Orders
- Sets
- Functions
- (Least) fixpoints and induction
- (Well-founded) recursion
- Arithmetic
- Datatypes

The Roadmap

We are still looking at how the different parts of mathematics are encoded in the Isabelle/HOL library.

- Orders
- Sets
- Functions
- (Least) fixpoints and induction
- (Well-founded) recursion
- Arithmetic
- Datatypes

Motivation

Current stage of our course:

- On the basis of conservative embeddings, set theory can be built safely.
- Inductive sets can be defined using least fixpoints and suitably supported by Isabelle.
- Well-founded orderings can be defined without referring to infinity. Recursive functions can be based on these. Needs inductive sets though. Support by Isabelle provided.

Motivation

Current stage of our course:

- On the basis of conservative embeddings, set theory can be built safely.
- Inductive sets can be defined using least fixpoints and suitably supported by Isabelle.
- Well-founded orderings can be defined without referring to infinity. Recursive functions can be based on these. Needs inductive sets though. Support by Isabelle provided.

Next important topic: arithmetic.

Which Approach to Take?

- Purely definitional?

Which Approach to Take?

- Purely definitional?

  Not possible with eight basic rules (cannot enforce infinity of HOL model)!

Which Approach to Take?

- Purely definitional?

  Not possible with eight basic rules (cannot enforce infinity of HOL model)!
- Heavily axiomatic? I.e., we state natural numbers by Peano axioms and claim analogous axioms for any other number type?

Which Approach to Take?

- Purely definitional?

  Not possible with eight basic rules (cannot enforce infinity of HOL model)!

- Heavily axiomatic? I.e., we state natural numbers by Peano axioms and claim analogous axioms for any other number type?

  Insecure!

Which Approach to Take?

- Purely definitional?

  Not possible with eight basic rules (cannot enforce infinity of HOL model)!

- Heavily axiomatic? I.e., we state natural numbers by Peano axioms and claim analogous axioms for any other number type?

  Insecure!

- Minimally axiomatic? We construct an infinite set, and define numbers etc. as inductive subset?

Which Approach to Take?

- Purely definitional?

  Not possible with eight basic rules (cannot enforce infinity of HOL model)!

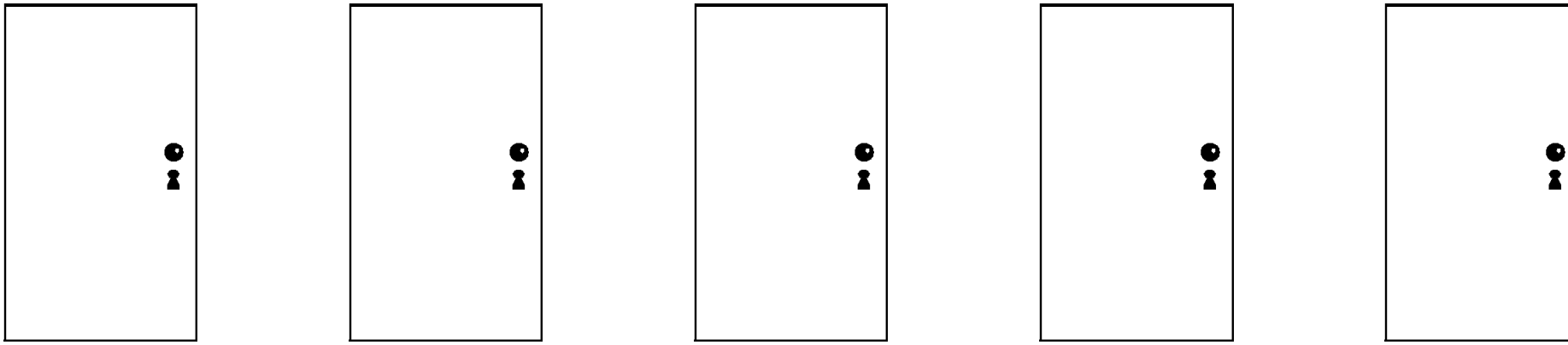
- Heavily axiomatic? I.e., we state natural numbers by Peano axioms and claim analogous axioms for any other number type?

  Insecure!

- Minimally axiomatic? We construct an infinite set, and define numbers etc. as inductive subset?

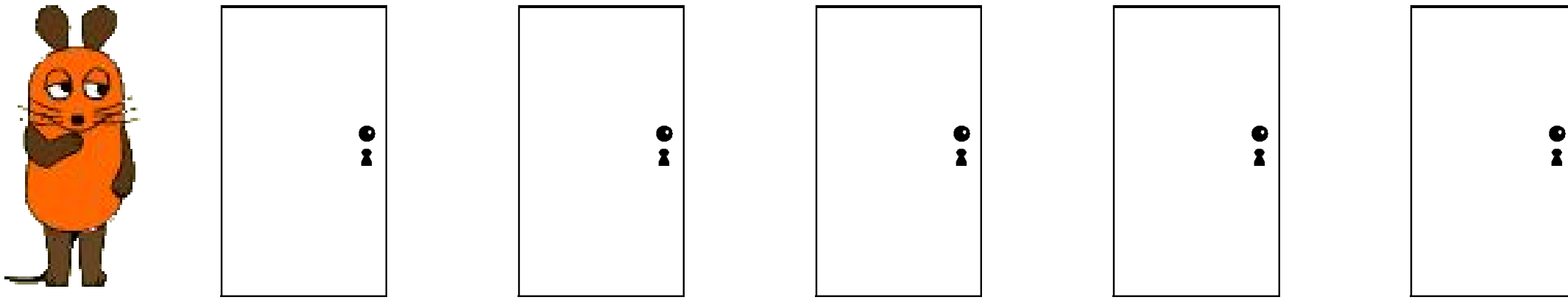
  Yes. Finally use infinity axiom.

What is Infinity? Cantor's Hotel



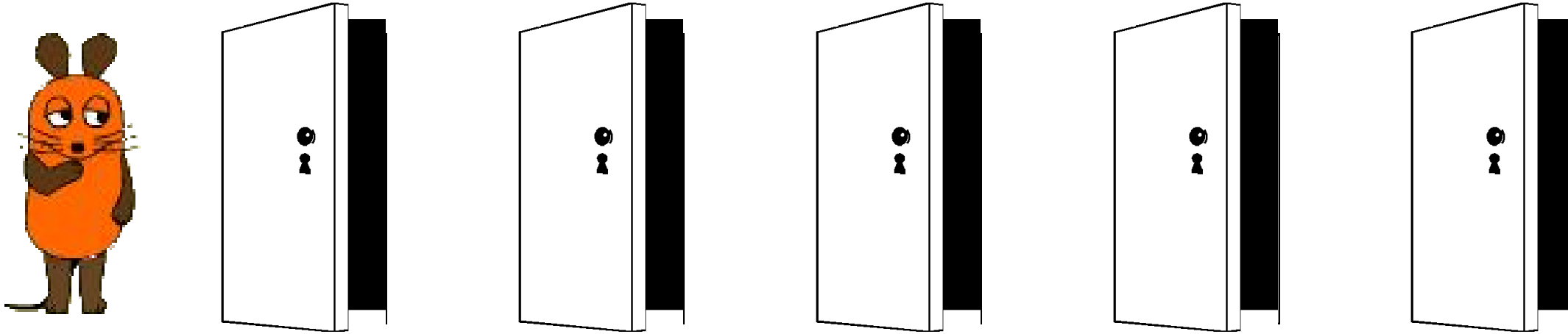
Cantor's hotel has infinitely many guests in his rooms if the receptionist can do the following procedure:

What is Infinity? Cantor's Hotel



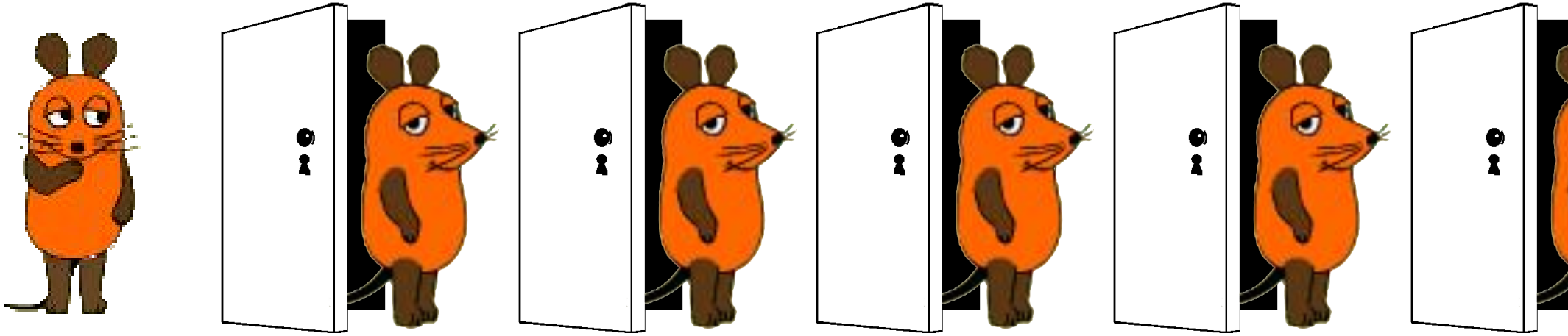
Cantor's hotel has infinitely many guests in his rooms if the receptionist can do the following procedure: A new guest arrives.

What is Infinity? Cantor's Hotel



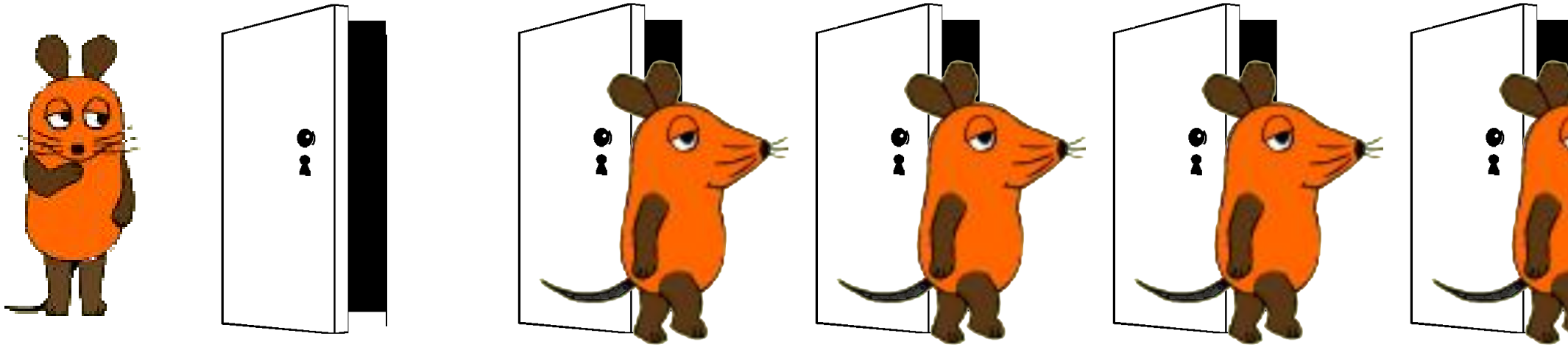
Cantor's hotel has infinitely many guests in his rooms if the receptionist can do the following procedure: A new guest arrives. The receptionist tells all guests

What is Infinity? Cantor's Hotel



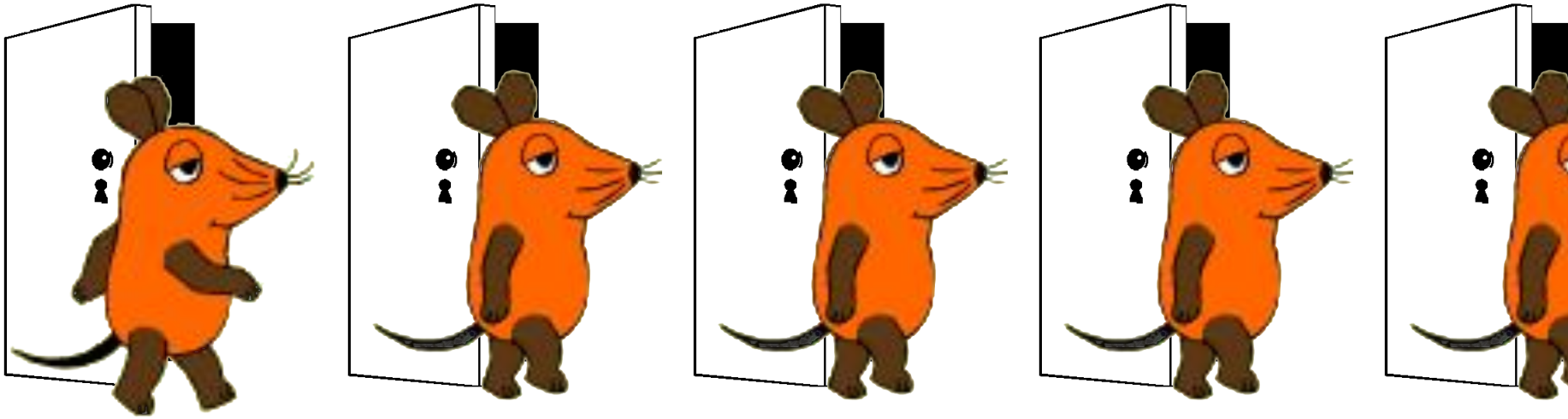
Cantor's hotel has infinitely many guests in his rooms if the receptionist can do the following procedure: A new guest arrives. The receptionist tells all guests to move one room.

What is Infinity? Cantor's Hotel



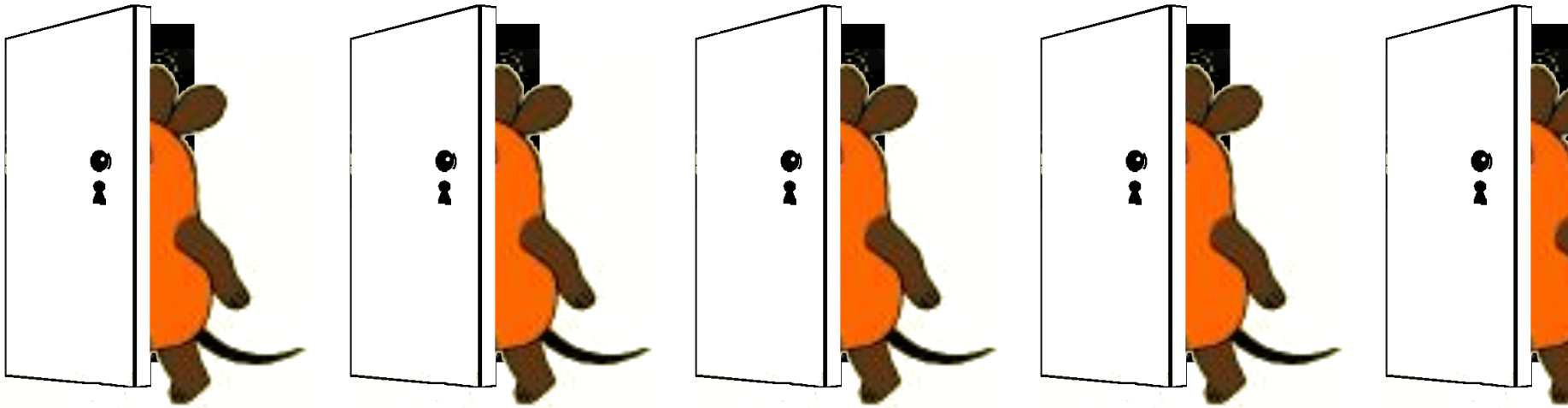
Cantor's hotel has infinitely many guests in his rooms if the receptionist can do the following procedure: A new guest arrives. The receptionist tells all guests to move one room. They move one room forward,

What is Infinity? Cantor's Hotel



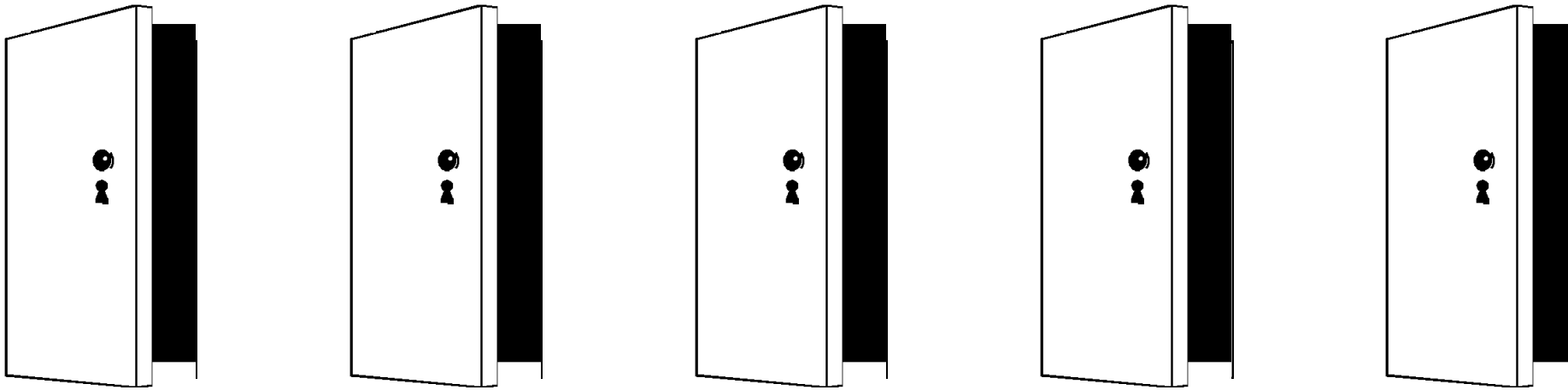
Cantor's hotel has infinitely many guests in his rooms if the receptionist can do the following procedure: A new guest arrives. The receptionist tells all guests to move one room. They move one room forward, the new guest takes the first room,

What is Infinity? Cantor's Hotel



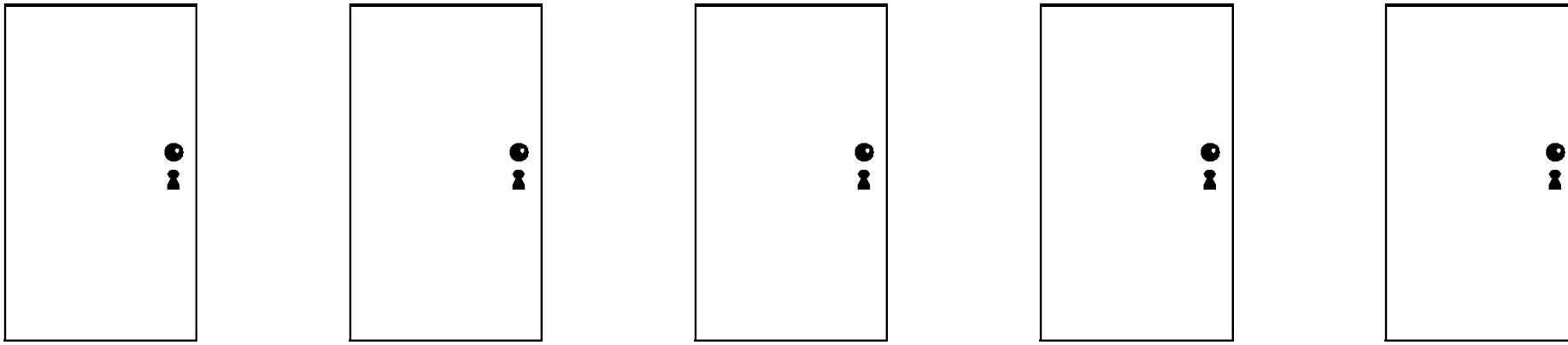
Cantor's hotel has infinitely many guests in his rooms if the receptionist can do the following procedure: A new guest arrives. The receptionist tells all guests to move one room. They move one room forward, the new guest takes the first room,

What is Infinity? Cantor's Hotel



Cantor's hotel has infinitely many guests in his rooms if the receptionist can do the following procedure: A new guest arrives. The receptionist tells all guests to move one room. They move one room forward, the new guest takes the first room,

What is Infinity? Cantor's Hotel



Cantor's hotel has infinitely many guests in his rooms if the receptionist can do the following procedure: A new guest arrives. The receptionist tells all guests to move one room. They move one room forward, the new guest takes the first room, and all are home and dry !

Axiom of Infinity

The axiomatic core of numbers:

Axiom of Infinity

The axiomatic core of numbers:

axioms infinity : "$\exists$ f :: ind $\Rightarrow$ ind . inj f $\wedge \neg$ surj f"

Axiom of Infinity

The axiomatic core of numbers:

axioms infinity : "$\exists$ f :: ind $\Rightarrow$ ind. inj f $\wedge \neg$ surj f"

where injective and surjective are:

inj f $\equiv \forall$ x. $\forall$ y. f(x)=f(y) $\rightarrow$ x=y
surj f $\equiv \forall$ y. $\exists$ x. y=f(x)

Axiom of Infinity

The axiomatic core of numbers:

axioms infinity : "$\exists$ f :: ind $\Rightarrow$ ind. inj f $\wedge \neg$ surj f"

where injective and surjective are:

inj f $\equiv \forall$ x. $\forall$ y. f(x)=f(y) $\rightarrow$ x=y
surj f $\equiv \forall$ y. $\exists$ x. y=f(x)

The axiom forces ind to be the "infinite type" (called "I" in [Chu40]).

Natural Numbers: Nat.thy

Based on the axiom of inifinity, a *proto*-Zero and a *proto*-Suc can be introduced by type specification:

consts
ZERO :: ind
SUC :: ind ⇒ ind

specification (SUC)
SUC_charn: inj SUC ∧¬ surj SUC
by (*rule* infinity)
specification (ZERO)
ZERO_charn: ZERO ≠SUC X
by (*insert* SUC_charn, auto simp: surj_def)

The proofs show that witnesses satisfy the required properties of the constants.

Defining the Set Nat

Now we define inductively a set generated by ZERO and SUC:

consts NAT :: ind set

inductive NAT
  intros
  ZERO_I: ZERO∈NAT
  SUC_I : ⟦ x∈NAT ⟧⟹SUC x∈NAT

(Recall that Isabelle converts this in:
$Nat = lfp\,(\lambda X.\{Zero_Rep\} \cup (Suc_Rep\,`\,X))$
and derives an induction scheme)

Defining the Type nat

The inductive set Nat is now abstracted via type definition to the type nat:

Defining the Type nat

The inductive set Nat is now abstracted via type definition to the type nat:

typedef (Nat)
  nat = "Nat" **by** (...)

Constants in nat

Moreover, we define 0 and Suc via their corresponding values in Nat :

consts

Suc :: nat $\Rightarrow$ nat

pred_nat :: (nat $\times$ nat) set

defs

Zero_nat_def: 0 $\equiv$ Abs_Nat Zero_Rep

Suc_def: Suc $\equiv$ (λn. Abs_Nat (Suc_Rep (Rep_Nat n)))

pred_nat_def: pred_nat $\equiv$ {(m, n). n = Suc m}

Some Theorems in Nat

From the induction inherited from Nat, we derive:

nat_induct $\llbracket\ P\ 0;\ \bigwedge n.P\ n \Longrightarrow P\ (Suc\ n)\ \rrbracket \Longrightarrow P\ n$

diff_induct $\llbracket\ \bigwedge x.\ P\ x\ 0;\ \bigwedge y.\ P\ 0\ (Suc\ y);$
$\bigwedge x\ y.P\ x\ y \Longrightarrow P\ (Suc\ x)(Suc\ y)\rrbracket$
$\Longrightarrow P\ m\ n$

Some Theorems in Nat

From the induction inherited from Nat, we derive:

$$\text{nat_induct} \quad [\![\ P\ 0;\ \bigwedge n.P\ n \Longrightarrow P\ (\text{Suc}\ n)\ ]\!] \ \Longrightarrow P\ n$$

$$\begin{aligned}\text{diff_induct} \quad & [\![\ \bigwedge x.\ P\ x\ 0;\ \bigwedge y.\ P\ 0\ (\text{Suc}\ y);\\ & \quad \bigwedge x\ y.P\ x\ y \Longrightarrow P\ (\text{Suc}\ x)(\text{Suc}\ y)]\!]\\ & \Longrightarrow P\ m\ n\end{aligned}$$

Moreover, we have as pre-requisite for wf-induction:

$$\text{wf}(\text{pred_nat})$$

These are the main weapons for proving theorems in basic number theory.

Nat.thy and Well-Founded Orders

Definition of orders:

Nat.thy and Well-Founded Orders

Definition of orders:

$$m < n \equiv (m, n) \in \mathsf{pred_nat}\hat{}+$$
$$m \leq (n :: \mathsf{nat}) \equiv \neg (n < m)$$

Nat.thy and Well-Founded Orders

Definition of orders:

$$m < n \equiv (m, n) \in \text{pred_nat}\hat{}+$$
$$m \leq (n :: \text{nat}) \equiv \neg (n < m)$$

have the properties:

$$m \leq m$$
$$[\![x \leq y;\ y \leq z]\!] \Longrightarrow x \leq z$$
$$[\![x \leq y;\ y \leq x]\!] \Longrightarrow x = y$$
$$x < y \vee y < x \vee x = y$$

Using Primitive Recursion

Nat.thy defines rich theory on nat. Uses **primrec** syntax for defining recursive functions, and case construct.

```
primrec
  add_0     0 + n = n
  add_Suc   Suc m + n = Suc(m + n)
primrec
  diff_0    m − 0 = m
  diff_Suc  m − Suc n = (case m − n of 0 => 0 | Suc k => k)
primrec
  mult_0    0 ∗ n = 0
  mult_Suc  Suc m ∗ n = n + (m ∗ n)
```

Some Theorems in Nat.thy

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	$m * n * k = m * (n * k)$
	$m * n = n * m$
	$x * (y * z) = y * (x * z)$

Some Theorems in Nat.thy

add_0_right	$m + 0 = m$
add_ac	$m + n + k = m + (n + k)$
	$m + n = n + m$
	$x + (y + z) = y + (x + z)$
mult_ac	$m * n * k = m * (n * k)$
	$m * n = n * m$
	$x * (y * z) = y * (x * z)$

Note third part of add_ac, mult_ac, respectively.

Some Theorems in Nat.thy

add_0_right	$m + 0 = m$
add_ac	$m + n + k = m + (n + k)$
	$m + n = n + m$
	$x + (y + z) = y + (x + z)$
mult_ac	$m * n * k = m * (n * k)$
	$m * n = n * m$
	$x * (y * z) = y * (x * z)$

Note third part of add_ac, mult_ac, respectively.

Technically, add_ac and mult_ac are lists of `thm`'s.

Proof of `add_0_right`

$$m + 0 = m \qquad add_0_right$$

Proof of `add_0_right`

$$n + 0 = n$$

$$\dfrac{\dfrac{\text{add_0}}{0 + 0 = 0} \qquad Suc\, n + 0 = Suc\, n}{m + 0 = m}\ \text{nat_induct}$$

Proof of `add_0_right`

$$\dfrac{\dfrac{\text{add_0}}{0+0=0} \qquad \dfrac{\dfrac{\dfrac{\text{add_suc}}{Suc\,m+n=Suc(m+n)}}{Suc(m+n)=Suc\,m+n}\,\textit{sym} \qquad \dfrac{n+0=n}{Suc(n+0)=Suc\,n}\,\textit{fun_cong}}{Suc\,n+0=Suc\,n}\,\textit{subst}}{m+0=m}\,\text{nat_induct}$$

Proof of `add_0_right`

$$\dfrac{\dfrac{\text{add_0}}{0+0=0} \qquad \dfrac{\dfrac{\dfrac{\text{add_suc}}{Suc\,m+n=Suc(m+n)}}{Suc(m+n)=Suc\,m+n}\,\textit{sym} \qquad \dfrac{[n+0=n]^1}{Suc(n+0)=Suc\,n}\,\textit{fun_cong}}{Suc\,n+0=Suc\,n}\,\textit{subst}}{m+0=m}\,\text{nat_induct}^1$$

Integers

The integers $\ldots, -2, -1, 0, 1, 2, \ldots$ are identified with equivalence classes over nat $\times$ nat (thought as "differences" $0 - 1, 1 - 2, 3 - 4, \ldots$).

Integers

The integers $\ldots, -2, -1, 0, 1, 2, \ldots$ are identified with equivalence classes over nat $\times$ nat (thought as "differences" $0-1, 1-2, 3-4, \ldots$).

IntDef = Equiv + NatArith +

constdefs

```
  intrel  :: ((nat×nat) ×(nat ×nat)) set
  intrel  ≡{p. ∃ x1 y1 x2 y2.
                          p=((x1::nat,y1),(x2,y2)) ∧
                          x1+y2 = x2+y1}
```

typedef (Integ)

```
  int = UNIV//intrel   (...)
```

Injections of nat's into integers, negation, addition, multiplication were now defined in terms of "differences":

int :: nat => int
int m ≡Abs_Integ(intrel " {(m,0)})

minus_int_def :
− z ≡Abs_Integ (⋃(x,y)∈Rep_Integ z. intrel "{(y,x)})

add_int_def :
z + w ≡ ...

add_int_def : z ∗ w ≡ ...

Injections of nat's into integers, negation, addition, multiplication were now defined in terms of "differences":

int :: nat => int
int m ≡Abs_Integ(intrel " {(m,0)})

minus_int_def :
− z ≡Abs_Integ (⋃(x,y)∈Rep_Integ z. intrel "{(y,x)})

add_int_def :
z + w ≡ ...

add_int_def : z ∗ w ≡ ...

Note that we use overloading here!!!

Some Theorems in `IntArith`

Some theorems on integers are:

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.

Further Number Theories

- Binary Integers (Bin.thy, for fast computation)
- Rational Numbers (HOL-Complex/Rational.thy)
- Real Numbers (HOL-Complex/Real.thy: based on Dedekind-sections of positive rationals.
- Hyperreals (HOL-Complex/Hyperreal.thy for non-standard analysis)
- Machine numbers such as JavaIntegers [RW04] and floats [Har98, Har00] for Intel's PentiumIV

Conclusion on Arithmetic

Using conservative extensions in HOL, we can build

- the naturals (as type definition based on ind), and
- higher number theories (via equivalence construction).

Conclusion on Arithmetic

Using conservative extensions in HOL, we can build

- the naturals (as type definition based on ind), and
- higher number theories (via equivalence construction).

Potential for

- analysis of processor arithmetic units, and
- function analysis in HOL (combination with computer algebra systems such as Mathematica).

Future: Analysis of hybrid systems.

The methodological overhead of the conservative method can be tackled by powerful mechanical support.

More Detailed Explanations

The Peano Axioms

The Peano axioms are:

- $0 \in nat$
- $\forall x.x \in nat \rightarrow Suc(x) \in nat$
- $\forall x.Suc(x) \neq 0$
- $\forall xy.Suc(x) = Suc(y) \rightarrow x = y$
- $\forall P.(P(0) \wedge \forall n.(P(n) \rightarrow P(Suc(n)))) \rightarrow \forall n.P(n)$

The latter formula is not an axiom in first-order logic, it is traditionally described as "axiom schema".

However, it fit's smoothely into HOL.

The case Statement for nat

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):

$$\texttt{case}\ z\ f\ n = \begin{cases} z & \text{if } n = 0 \\ f\ k & \text{if } n = Suc\ k \end{cases}$$

An ML-like pattern match construct in:

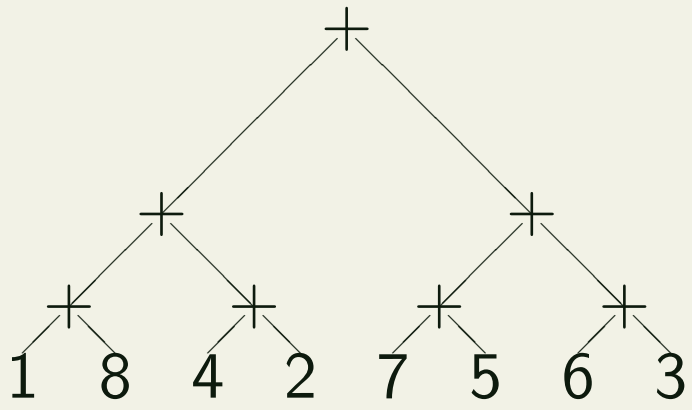
diff_Suc "m − Suc n = (case m − n **of** 0 => 0 | Suc k => k)"

uses a paraphrasing for case 0 (λ x.x) (n−m).

Left Commutation

The theorems $x + (y + z) = y + (x + z)$ and $x * (y * z) = y * (x * z)$ are called left-commutation laws and are crucial for (ordered) rewriting.

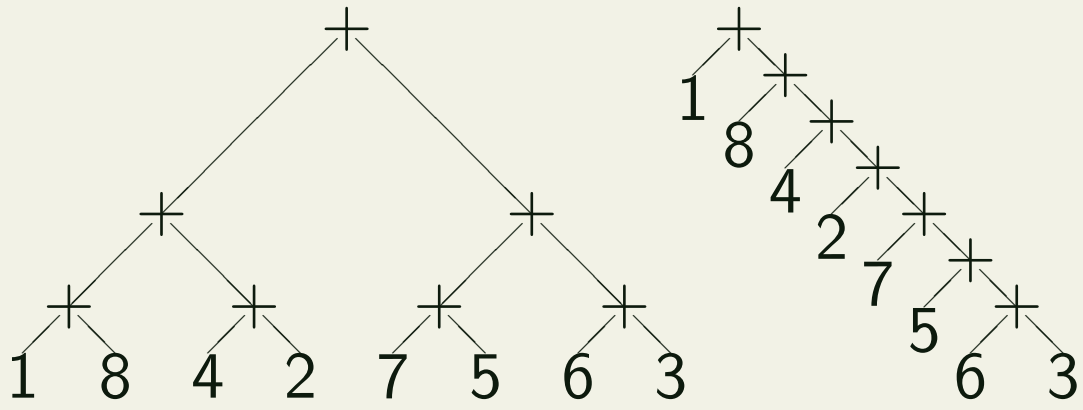
Suppose we have the term shown below.



Left Commutation

The theorems $x + (y + z) = y + (x + z)$ and $x * (y * z) = y * (x * z)$ are called left-commutation laws and are crucial for (ordered) rewriting.

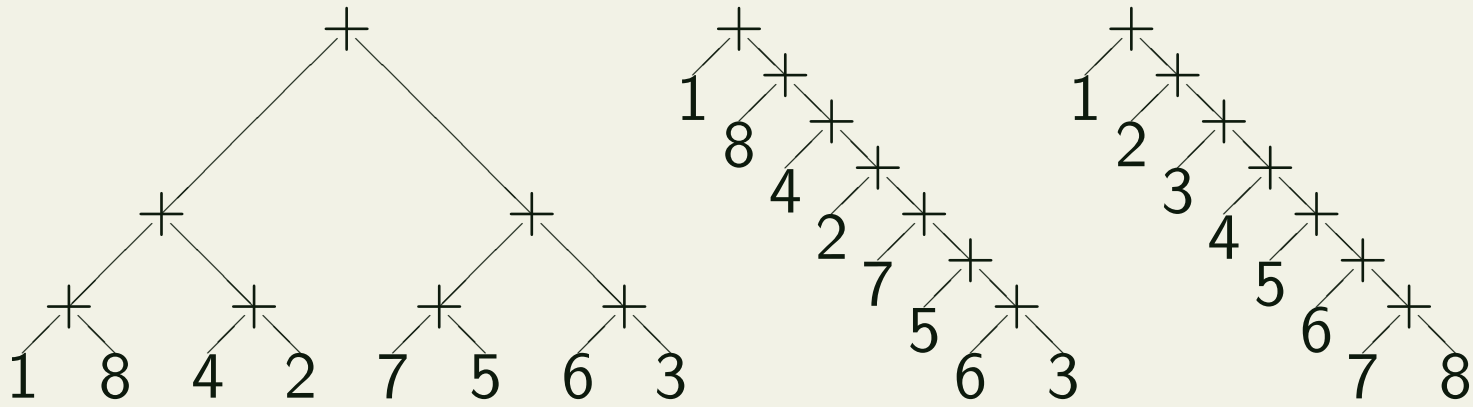
Suppose we have the term shown below. Using associativity $(m + n + k = m + (n + k))$ this will be rewritten to the second term.



Left Commutation

The theorems $x + (y + z) = y + (x + z)$ and $x * (y * z) = y * (x * z)$ are called left-commutation laws and are crucial for (ordered) rewriting.

Suppose we have the term shown below. Using associativity $(m + n + k = m + (n + k))$ this will be rewritten to the second term. Using left-commutation, this will be rewritten to the third term. This is a so-called AC-normal form, for an appropriately chosen term ordering.



Equivalence Classes

Recall the general concept of an equivalence relation. Generally, for a set S and an equivalence relation R defined on the set, one can define $S//R$, the quotient of S w.r.t. R.

$$S//R = \{A \mid A \subseteq S \wedge \forall x, y \in A.(x, y) \in R\}$$

That is, one partitions the set S into subsets such that each subset collects equivalent elements. This is a mathematical standard concept.

We explain it for integers in more detail. One can view a pair (n, m) of natural numbers as representation of the integer $n - m$. But then (n, m) and (n', m') represent the same integer if and only if $n - m = n' - m'$, or equivalently, $n + m' = n' + m$. In this case (n, m) and (n', m') are said to be equivalent. The set of equivalent elements is an equivalence class. The quotient maps therefore a set to a set of equivalence classes.

Reals According to Dedekind

The reals have been axiomatized by Dedekind by stating that a set R is partitioned into two sets A and B such that $R = A \cup B$ and for all $a \in A$ and $b \in B$, we have $a < b$. Now there is a number s such that $a \leq s \leq b$ for all $a \in A$ and $b \in B$. The irrational numbers are characterised by the fact that there exists exactly one such s. This axiomatization has been used as a basis for formalizing real numbers in Isabelle/HOL.

Hyperreals

In non-standard analysis, one works with sequences that are not necessarily converging. This is a relatively new field in mathematics and Isabelle/HOL has been successfully applied in it [FP98]. We just mention this here to say that Isabelle/HOL is used for "cutting-edge" mathematics and not just toy examples.

Hybrid Systems

Hybrid systems is a field in software engineering concerned with using finite automata for controlling physical systems such as ABS in cars etc.

Higher-Order Logic: Datatypes

Burkhart Wolff

The Roadmap

We are still looking at how the different parts of mathematics are encoded in the Isabelle/HOL library.

- Orders
- Sets
- Functions
- (Least) fixpoints and induction
- (Well-founded) recursion
- Arithmetic
- Datatypes

The Roadmap

We are still looking at how the different parts of mathematics are encoded in the Isabelle/HOL library.

- Orders
- Sets
- Functions
- (Least) fixpoints and induction
- (Well-founded) recursion
- Arithmetic
- Datatypes

Datatypes: Motivation

Last lecture: Construction of natural numbers.

How can we build "datatypes" as conservative extension?

Datatypes: Motivation

Last lecture: Construction of natural numbers.

How can we build "datatypes" as conservative extension?

Can we generalize the natural number construction to arbitrary datatypes?

Datatypes: Motivation

Last lecture: Construction of natural numbers.

How can we build "datatypes" as conservative extension?

Can we generalize the natural number construction to arbitrary datatypes?

- Well, yes — by Gödelization . . .

Datatypes: Motivation

Last lecture: Construction of natural numbers.

How can we build "datatypes" as conservative extension?

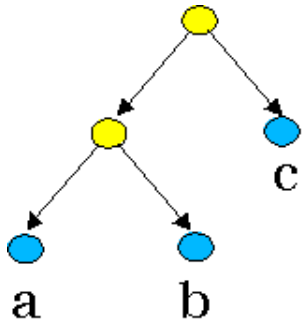
Can we generalize the natural number construction to arbitrary datatypes?

- Well, yes — by Gödelization . . .
- . . . and by an S-expressions-like tree data structure and inductive definitions.

Caveat: We will simplify! See Datatype_Universe.thy and [Wen99] for Details.

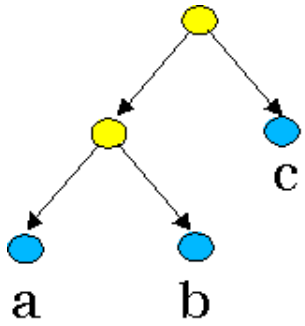
S-Expressions

Idea: We build an "ancestor"-datatype of binary trees $\alpha\ dtree$ (LISP-like S-expressions).



S-Expressions

Idea: We build an "ancestor"-datatype of binary trees $\alpha\ dtree$ (LISP-like S-expressions).



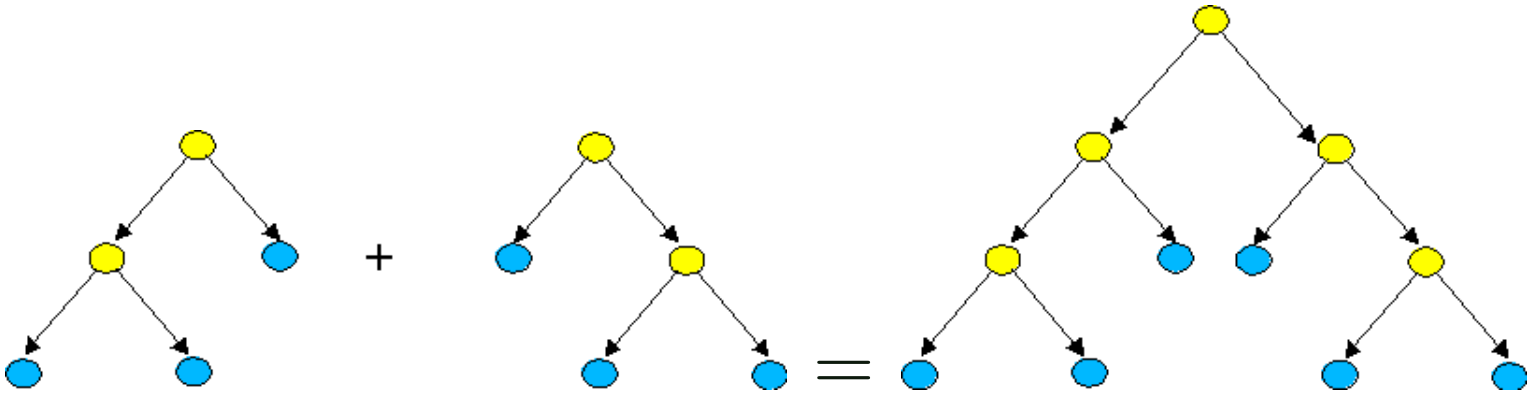
This is encoded as a set of "nodes" (defined by their path from the root and a value in the leaves), e.g.:

$$\{(\langle 0,0\rangle, a), (\langle 0,1\rangle, b), (\langle 1\rangle, c)\}$$

Building Trees

- $\texttt{Atom}(n)$

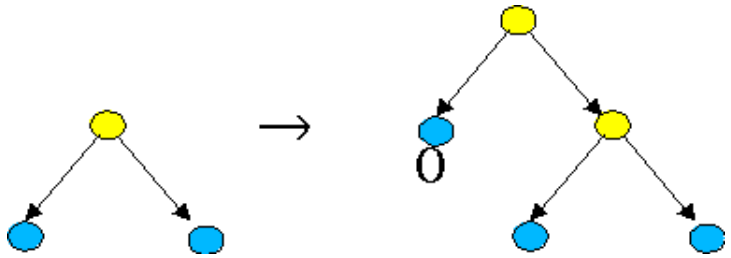
- $\texttt{Scons}\ X\ Y$



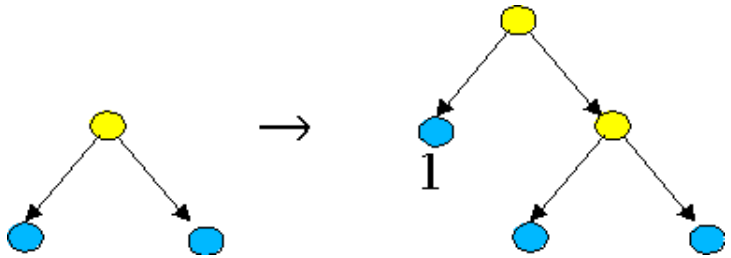
Tagging Trees

We want to tag an S-expression by either 0 or 1. This can be done by "`Scons`"-ing it with an S-expression consisting of an administration label. By convention, the tag is to the left.

- `In0_def` $\quad In0(X) \equiv Scons\ Atom(Inr(0))(X)$



- `In1_def` $\quad In1(X) \equiv Scons\ Atom(Inr(1))(X)$



Products and Sums on Sets of S-Expressions

Product of two sets A and B of S-expressions: All `Scons`-trees where left subtree from A, right subtree from B.

`uprod_def` $$uprod\ A\ B \equiv \bigcup_{x \in A} \bigcup_{y \in B} \{(Scons\ x\ y)\}$$

Products and Sums on Sets of S-Expressions

Product of two sets A and B of S-expressions: All `Scons`-trees where left subtree from A, right subtree from B.

$$\texttt{uprod_def} \quad uprod\, A\, B \equiv \bigcup_{x \in A} \bigcup_{y \in B} \{(Scons\, x\, y)\}$$

Sum of two sets A and B of S-expressions: union of A and B after S-expressions in A have been tagged 0 and S-expressions in B have been tagged 1, so that one can tell where they come from.

$$\texttt{usum_def} \quad usum\, A\, B \equiv In0\, `\, A \cup In1\, `\, B$$

Some Properties of Trees and Tree Sets

- `Atom`, `In0`, `In1`, `Scons` are injective.
- `Atom` and `Scons` are pairwise distinct. `In0` are `In1` pairwise distinct.

Some Properties of Trees and Tree Sets

- `Atom`, `In0`, `In1`, `Scons` are injective.
- `Atom` and `Scons` are pairwise distinct. `In0` are `In1` pairwise distinct.
- Tree sets represent a universe that is closed under products and sums: `usum`, `uprod` have type $[(\alpha\ dtree)\ set, (\alpha\ dtree)\ set] \Rightarrow (\alpha\ dtree)\ set$.
- `uprod` and `usum` are monotone.

Some Properties of Trees and Tree Sets

- `Atom`, `In0`, `In1`, `Scons` are injective.
- `Atom` and `Scons` are pairwise distinct. `In0` are `In1` pairwise distinct.
- Tree sets represent a universe that is closed under products and sums: `usum`, `uprod` have type $[(\alpha\ dtree)\ set, (\alpha\ dtree)\ set] \Rightarrow (\alpha\ dtree)\ set$.
- `uprod` and `usum` are monotone.
- Tree sets represent a universe that is closed under products and sums combined with arbitrary applications of lfp.

Reminder: we simplified!

Lists in Isabelle

Now we define inductively a subset of S-expressions having the "structure of lists". As a pre-requisite, we define "raw constructors":

constdefs

```
NIL :: 'a dtree
"NIL ≡In0(Atom(Inr(0)))"
CONS :: ['a dtree, 'a dtree] ⇒ 'a dtree
"CONS M N ≡In1(Scons M N)"
```

Lists as S-Expressions: Intuition

Examples of how lists would be represented as S-expressions:

Nil	$[]$
$Cons(7, Nil)$	$[7]$
$Cons(5, Cons(7, Nil))$	$[5, 7]$

Lists as S-Expressions: Intuition

Examples of how lists would be represented as S-expressions:

Nil	$[]$
$In0(Atom(Inr(0)))$	
$Cons(7, Nil)$	$[7]$
$Cons(5, Cons(7, Nil))$	$[5, 7]$

Lists as S-Expressions: Intuition

Examples of how lists would be represented as S-expressions:

Nil	$[]$
$In0(Atom(Inr(0)))$	
$Cons(7, Nil)$	$[7]$
$CONS\ (Atom(Inl\,7))\ In0(Atom(Inr(0)))$	
$Cons(5, Cons(7, Nil))$	$[5, 7]$

Lists as S-Expressions: Intuition

Examples of how lists would be represented as S-expressions:

Nil	$[]$
$\quad In0(Atom(Inr(0)))$	
$Cons(7, Nil)$	$[7]$
$\quad CONS\ (Atom(Inl\,7))\ In0(Atom(Inr(0)))$	
$Cons(5, Cons(7, Nil))$	$[5, 7]$
$\quad CONS\ (Atom(Inl\,5))$	
$\quad\quad (CONS\ (Atom(Inl\,7))\ In0(Atom(Inr(0))))$	

Now let's construct the S-expressions having this form.

Lists as S-Expressions: Inductive Construction

Based on the "raw constructors", we define the inductive set of S-expressions:

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 `src/HOL/Induct/SList.thy` in the Isabelle distribution for details!

Defining the "Real" List Type

We apply a type definition in order to define the type list by the inductive subset list (A).

Defining the "Real" List Type

We apply a type definition in order to define the type list by the inductive subset list (A).

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 $t :: \alpha$.

This is an example of a definition of a polymorphic type.

List Constructors

We define the real constructor names for lists:

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))"
```

. . . derive the induction scheme and forget about NIL and $CONS$.

Summary

Similar to primitive recursion, compilers for datatype definitions can be provided that preform a conservative construction behind the scene

Summary

Similar to primitive recursion, compilers for datatype definitions can be provided that preform a conservative construction behind the scene

datatype 'a list = Nil | Cons 'a ('a list)

Summary

Similar to primitive recursion, compilers for datatype definitions can be provided that preform a conservative construction behind the scene

```
datatype 'a list = Nil | Cons 'a ('a list)
```

In particular, this automates the proofs of:

- the induction theorem;
- distinctness;
- injectivity of constructors.

Summary

Similar to primitive recursion, compilers for datatype definitions can be provided that preform a conservative construction behind the scene

```
datatype 'a list = Nil | Cons 'a ('a list)
```

In particular, this automates the proofs of:

- the induction theorem;
- distinctness;
- injectivity of constructors.

Summary

Similar to primitive recursion, compilers for datatype definitions can be provided that preform a conservative construction behind the scene

```
datatype 'a list = Nil | Cons 'a ('a list)
```

In particular, this automates the proofs of:

- the induction theorem;
- distinctness;
- injectivity of constructors.

This also works for mutually and indirectly recursive datatype definitions.

More Detailed Explanations

Gödelization

In computation theory, Gödelization is the process of encoding data structures (words, trees, . . .) into natural numbers.

Assume that we want to encode "binary tree's" over natural numbers. These tree's could be defined as solutions of the following set equation:

$$S(Nat) = Nat + (S(Nat) \times S(Nat))$$

or by the corresponding free data type:

```
datatype S = Atom nat
           | Node [S, S] => S
```

This notation implies that

1. Atom and Node produce distinct values,
2. Atom and Node are injective

3. all values in S are inductively generated over Atom and Node.

Can we have a model satisfying these requirements so far?

The answer is yes: Consider $Atom\,x = 2^x$ and $Node\,x\,y = 3^x * 5^y$. Distinctness holds obviously, and due to uniqueness of prime number factorization, both functions are injective. Building an inductive subset of nat generated by Atom and Node will also give the induction principle.

Unfortunately, the construction is not polymorphic as the presented S-expression-construction.

S-Expressions Explained

The datastructure we have in mind here consists of binary trees where the inner nodes are not labeled, and the leaves are labeled

- either with a term of arbitrary type, in which case the leaf would be an actual "piece of content" in the datastructure,
- or with a natural number, in which case the leaf serves special purposes for organizing our datastructure, as we will see later.

I.e., such binary trees have a type parametrized by a type variable α, the type of the latter kind of leaves. Let us call the type of such trees $\alpha\ dtree$.

As always with parametric polymorphism, when we consider how the datastructure as such works, we are not interested in what the values in the former kind of leaves are. This is just like the type and values of list elements are irrelevant for concatenating two lists. Of course, α could,

by coincidence, be instantiated to type `nat`.

Think of a label of the first kind as content label and a label of the second kind as administration label.

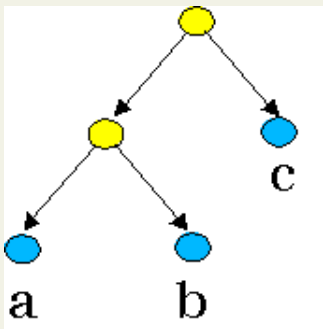
Technically, if something is either of this type or of that type, we are talking about a sum type. So a leaf label has type $\alpha + \texttt{nat}$ (written $(\alpha, \texttt{nat})\ \texttt{sum}$ before), and it has the form either $\texttt{Inl}(a)$ for some $a :: \alpha$, or $\texttt{Inr}(n)$ for some $n :: \texttt{nat}$.

Path Sets Explained

The set

$$\{(\langle 0,0\rangle, a), (\langle 0,1\rangle, b), (\langle 1\rangle, c)\}$$

represents the tree



The path $\langle 0,0\rangle$ means: from the root take left subtree, then again left subtree. The path $\langle 1\rangle$ means: take right subtree.

How can a path $\langle p_0, \ldots, p_n\rangle$ be represented? One idea is to use the

function $f :: \texttt{nat} \Rightarrow \texttt{nat}$ defined by

$$f\ i = \begin{cases} p_i & \text{if } i \leq n \\ 2 & \text{otherwise} \end{cases}$$

as representation of $\langle p_0, \ldots, p_n \rangle$.

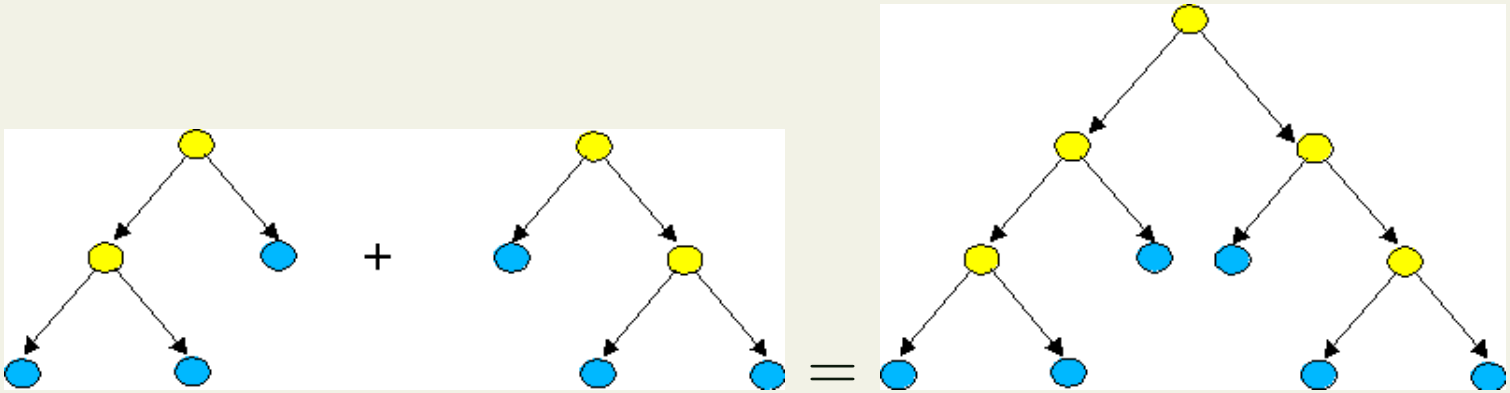
Atom

`Atom` takes a leaf label and turns it into a (simplest possible) S-expression (tree).

So it has type $\alpha + nat \Rightarrow \alpha\ dtree$.

Scons

Scons takes two S-expressions and creates a new S-expression as illustrated below:



So it has type $[\alpha\ dtree, \alpha\ dtree] \Rightarrow \alpha\ dtree$.

$In0 \text{`} \ldots, In1 \text{`} \ldots$

Recall that ‘ denotes the image of a function applied to a set.

Injective and Pairwise Distinct Functions

This means that any of $Atom$, $In0$, $In1$, $Scons$ applied to different S-expressions will return different S-expressions.

Moreover, a term with root $Scons$ is definitely different from a term with root $Atom$, and a term with root $In0$ is definitely different from a term with root $In1$.

Why is this important? It is an inherent characteristic of a datatype. A datatype consists of terms constructed using term constructors and is uinquely defined by what it is syntactically (one also says that terms are generated freely using the constructors). For example, injectivity of Suc and pairwise-distinctness of 0 and Suc mean for any two numbers m and n, the terms $\underbrace{Suc(\ldots Suc}_{m \text{ times}}(0)\ldots)$ and $\underbrace{Suc(\ldots Suc}_{n \text{ times}}(0)\ldots)$ are different.

Computing the Closure

Given a set T of trees (S-expressions), the closure of T under `Atom`, `In0`, `In1`, `Scons`, `usum`, `uprod` is the smallest set T' such that $T \subseteq T'$ and given any tree (or two trees, as applicable) from T', any tree constructable using `Atom`, `In0`, `In1`, `Scons`, `usum`, `uprod` is also contained in T'.

Remembering the construction of inductivelty defined sets, the closure is the least fixpoint of a monotone function adding trees to a tree set. This function must be constructed using `Atom`, `In0`, `In1`, `Scons`, `usum`, `uprod`. We do not go into the details, but note that it is crucial that `uprod` and `usum` are monotone, and note as well that slight complications arise from the fact that `usum` and `uprod` have type $[(\alpha\ dtree)\ set, (\alpha\ dtree)\ set] \Rightarrow (\alpha\ dtree)\ set$ rather than $(\alpha\ dtree)\ set \Rightarrow (\alpha\ dtree)\ set$.

The Expected Type of $Cons$

$Cons$ 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-expression) and a "list element", we must first wrap the list element by $Atom \circ Inl$, so that it becomes an s-expression.

List Syntaxes

Nil, $Cons(7, Nil)$, $Cons(5, Cons(7, Nil))$ are lists written according to what some programming languages introduce as the first, "official" syntax for lists.

For convenience, programming languages typically allow for the same lists to be written as $[]$, $[7]$, $[5, 7]$.

Higher-Order Logic Applications: IMP

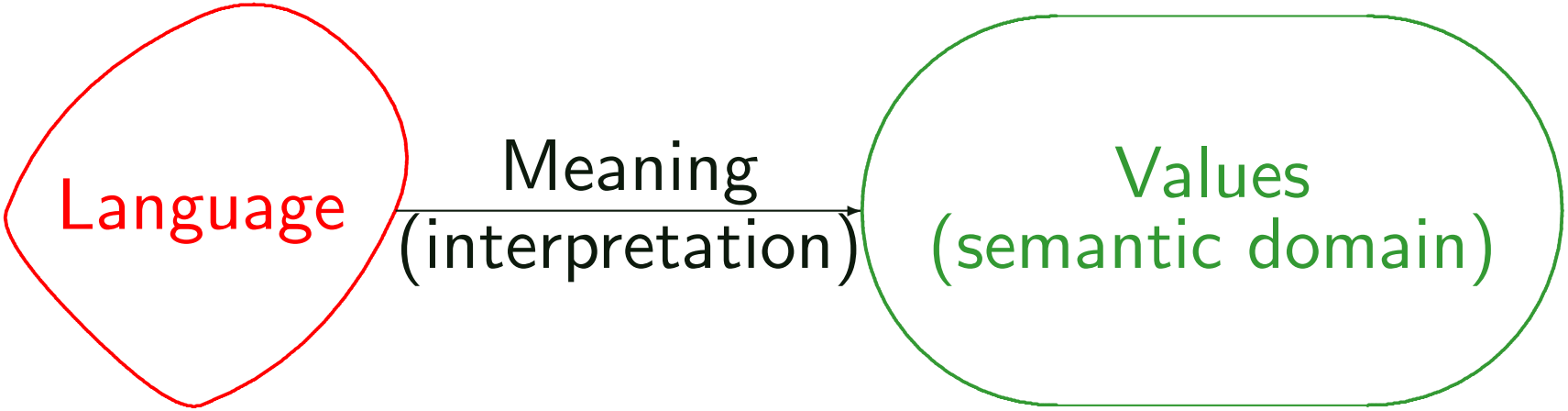
Jan Smaus and Burkhart Wolff

Language Semantics: An Introduction

Question: What is the meaning of a (programming or specification) language?

Language Semantics: An Introduction

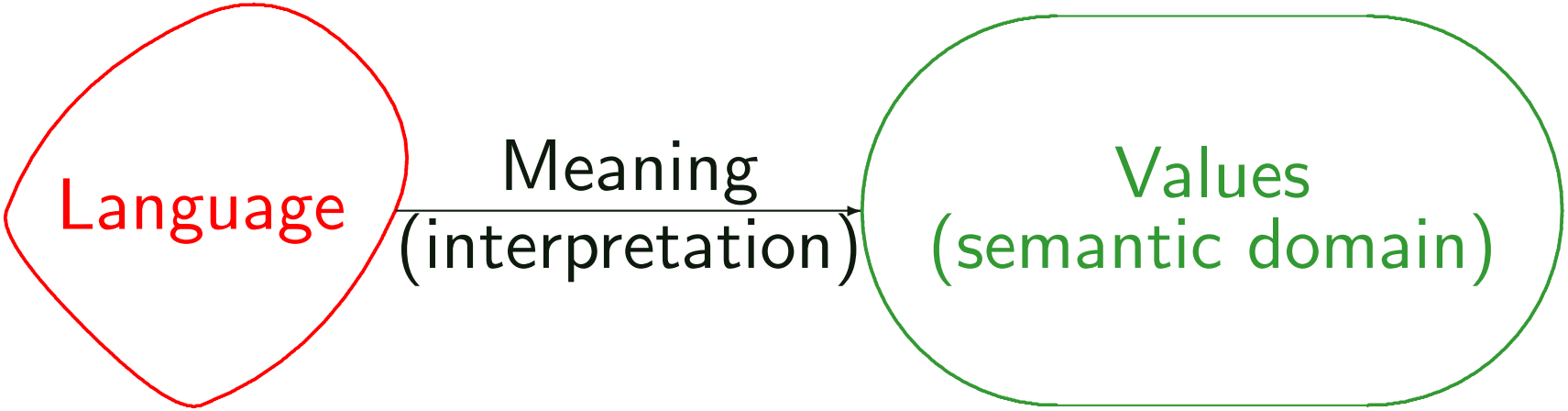
Question: What is the meaning of a (programming or specification) language?



- Syntax: language = set of symbols

Language Semantics: An Introduction

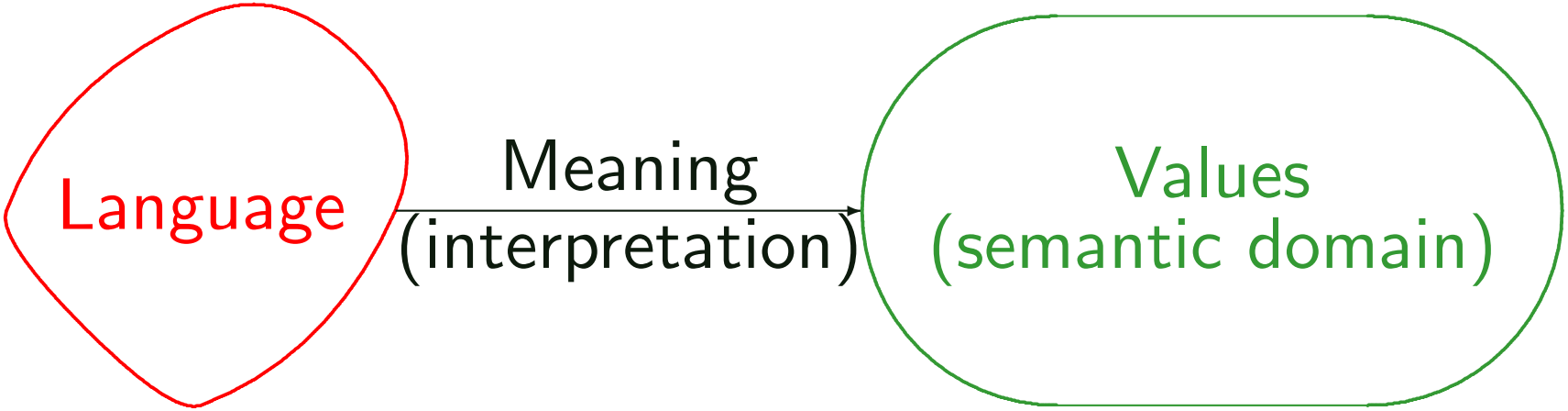
Question: What is the meaning of a (programming or specification) language?



- Syntax: language = set of symbols
- Semantics: set of denotations, the semantic domain

Language Semantics: An Introduction

Question: What is the meaning of a (programming or specification) language?



- Syntax: language = set of symbols
- Semantics: set of denotations, the semantic domain
- Meaning is a function (interpretation Sem) relating them.

Terminology

- embedding:

Terminology

- embedding: a theory representing syntax and semantics of a language in a theorem prover.

Terminology

- embedding: a theory representing syntax and semantics of a language in a theorem prover.
- object-language

Terminology

- embedding: a theory representing syntax and semantics of a language in a theorem prover.
- object-language is the language to be represented,

Terminology

- embedding: a theory representing syntax and semantics of a language in a theorem prover.
- object-language is the language to be represented,meta-language

Terminology

- embedding: a theory representing syntax and semantics of a language in a theorem prover.
- object-language is the language to be represented,meta-language the language used for this (e.g. Pure for HOL)

Terminology

- embedding: a theory representing syntax and semantics of a language in a theorem prover.
- object-language is the language to be represented,meta-language the language used for this (e.g. Pure for HOL)
- a deep embedding

Terminology

- embedding: a theory representing syntax and semantics of a language in a theorem prover.
- object-language is the language to be represented,meta-language the language used for this (e.g. Pure for HOL)
- a deep embedding declares syntax as a data type and defines an explicit interpretation Sem

Terminology

- embedding: a theory representing syntax and semantics of a language in a theorem prover.
- object-language is the language to be represented,meta-language the language used for this (e.g. Pure for HOL)
- a deep embedding declares syntax as a data type and defines an explicit interpretation Sem
- a shallow embedding

Terminology

- embedding: a theory representing syntax and semantics of a language in a theorem prover.
- object-language is the language to be represented,meta-language the language used for this (e.g. Pure for HOL)
- a deep embedding declares syntax as a data type and defines an explicit interpretation Sem
- a shallow embedding just provides the semantic functions for the operations of the language (no own syntax; e.g. variables are represented by meta-language variables).

Imperative Languages in the Isabelle/HOL Library

There are several embeddings of imperative languages in Isabelle/HOL [Nip02]:

- Hoare
- IMP
- IMPP
- MicroJava

Imperative Languages in the Isabelle/HOL Library

There are several embeddings of imperative languages in Isabelle/HOL [Nip02]:

- Hoare: shallowish, good examples
- IMP
- IMPP
- MicroJava

Imperative Languages in the Isabelle/HOL Library

There are several embeddings of imperative languages in Isabelle/HOL [Nip02]:

- Hoare: shallowish, good examples
- IMP: deepish, good theory
- IMPP
- MicroJava

Imperative Languages in the Isabelle/HOL Library

There are several embeddings of imperative languages in Isabelle/HOL [Nip02]:

- Hoare: shallowish, good examples
- IMP: deepish, good theory
- IMPP: extends IMP with procedures
- MicroJava

Imperative Languages in the Isabelle/HOL Library

There are several embeddings of imperative languages in Isabelle/HOL [Nip02]:

- Hoare: shallowish, good examples
- IMP: deepish, good theory
- IMPP: extends IMP with procedures
- MicroJava: deep, complex, powerful, state-of-the-art

Imperative Languages in the Isabelle/HOL Library

There are several embeddings of imperative languages in Isabelle/HOL [Nip02]:

- Hoare: shallowish, good examples
- IMP: deepish, good theory
- IMPP: extends IMP with procedures
- MicroJava: deep, complex, powerful, state-of-the-art

We choose IMP to learn a bit about "good ole imperative languages".

IMP offers:

- operational semantics;

IMP offers:

- operational semantics;
  - natural semantics[Plo81, CDTK86];
  - transition semantics[Plo81];

IMP offers:

- operational semantics;
  - natural semantics[Plo81, CDTK86];
  - transition semantics[Plo81];

- denotational semantics;

IMP offers:

- operational semantics;
  - natural semantics[Plo81, CDTK86];
  - transition semantics[Plo81];

- denotational semantics;
- axiomatic semantics (Hoare logic);

IMP offers:

- operational semantics;
  - natural semantics[Plo81, CDTK86];
  - transition semantics[Plo81];

- denotational semantics;
- axiomatic semantics (Hoare logic);
- equivalence proofs;

IMP offers:

- operational semantics;
  - natural semantics[Plo81, CDTK86];
  - transition semantics[Plo81];

- denotational semantics;
- axiomatic semantics (Hoare logic);
- equivalence proofs;
- weakest preconditions and verification condition generator.

IMP offers:

- operational semantics;
  - natural semantics[Plo81, CDTK86];
  - transition semantics[Plo81];

- denotational semantics;
- axiomatic semantics (Hoare logic);
- equivalence proofs;
- weakest preconditions and verification condition generator.

It closely follows the standard textbook [Win96].

The Command Language (Syntax)

The (abstract) syntax is defined in Com.thy.

The Command Language (Syntax)

The (abstract) syntax is defined in Com.thy.

```
Com = Main +
typedecl loc
types
  val   = nat (*arb.*)
  state = loc⇒val
  aexp  = state⇒val
  bexp  = state⇒bool
```

```
datatype com =
  SKIP
  | ":==" loc aexp (infixl 60)
  | Semi com com ("_ ; _" [60, 60]10)
  | Cond bexp com com
                ("IF _ THEN _ ELSE _"60)
  | While bexp com ("WHILE _ DO _"60)
```

The Command Language (Syntax)

The (abstract) syntax is defined in Com.thy.

```
Com = Main +
typedecl loc
types
  val   = nat (*arb.*)
  state = loc⇒val
  aexp  = state⇒val
  bexp  = state⇒bool
```

```
datatype com =
  SKIP
  | ":==" loc aexp (infixl 60)
  | Semi com com ("_ ; _" [60, 60]10)
  | Cond bexp com com
                  (" IF _ THEN _ ELSE _"60)
  | While bexp com ("WHILE _ DO _"60)
```

The type loc stands for locations. Note expressions are represented using the shallow technique. The datatype *com* stands for commands (command sequences).

Operational Semantics: Two Kinds

Natural semantics

Operational Semantics: Two Kinds

Natural semantics (idea: a program relates states):

Operational Semantics: Two Kinds

Natural semantics (idea: a program relates states):

evalc :: (com × state × state) set

Operational Semantics: Two Kinds

Natural semantics (idea: a program relates states):

evalc :: (com ×state × state) set

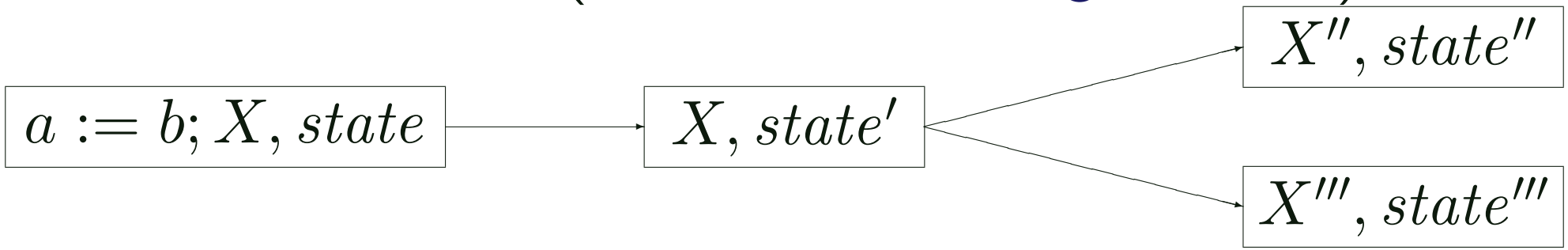
Transition semantics

Operational Semantics: Two Kinds

Natural semantics (idea: a program relates states):

evalc :: (com ×state × state) set

Transition semantics (idea: relates "configurations"):

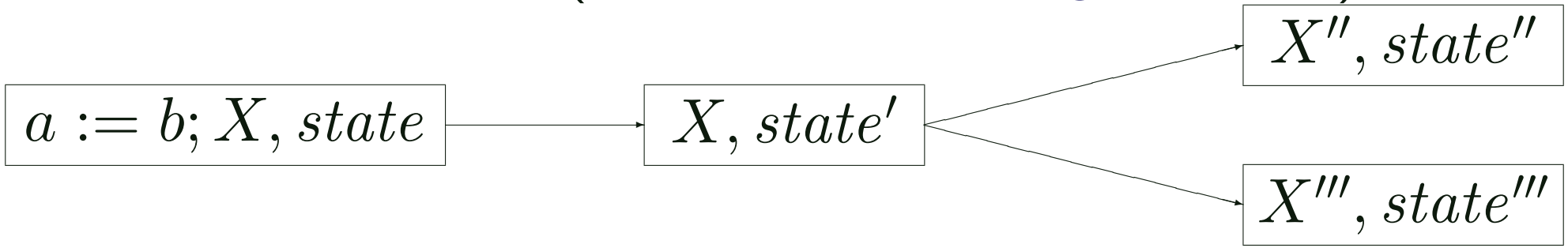


Operational Semantics: Two Kinds

Natural semantics (idea: a program relates states):

evalc :: (com ×state × state) set

Transition semantics (idea: relates "configurations"):



evalc1 :: ((com ×state) × (com ×state)) set

Natural Semantics

The transition relation of natural semantics is inductively defined.

Natural Semantics

The transition relation of natural semantics is inductively defined.

This means intuitively: The given steps are the only steps.

Natural Semantics

The transition relation of natural semantics is inductively defined.

This means intuitively: The given steps are the only steps.

consts evalc :: (com ×state × state) set
translations "<cm,s> −c−> s' " ≡"(cm,s,s') ∈ evalc"

Natural Semantics

The transition relation of natural semantics is inductively defined.

This means intuitively: The given steps are the only steps.

consts evalc :: (com ×state × state) set
translations "<cm,s> −c−> s' " ≡"(cm,s,s') ∈ evalc"

We now start giving the actual inductive definition of the natural semantics transition relation . . .

Natural Semantics: Skip and Assignment

inductive evalc

intrs

Skip: ⟨SKIP,s⟩ −c−> s

Assign ⟨x :== a,s⟩ −c−> s[x::=(a s)]

Natural Semantics: Skip and Assignment

inductive evalc

intrs

Skip: ⟨SKIP,s⟩ −c−> s

Assign ⟨x :== a,s⟩ −c−> s[x::=(a s)]

Note that s[x::=(a s)] is an abbreviation for update s x (a s), where

update s x v ≡ λ y. if y=x then v else s y

Natural Semantics: Skip and Assignment

```
inductive evalc
  intrs
  Skip:   ⟨SKIP,s⟩ −c−> s
  Assign  ⟨x :== a,s⟩ −c−> s[x::=(a s)]
```

Note that s[x::=(a s)] is an abbreviation for update s x (a s), where

update s x v ≡ λ y. if y=x then v else s y

Note that a is of type aexp or bexp.

Natural Semantics: Sequential Composition

Semi ⟦ <c_0,s> −c−> s_1; <c_1,s_1> −c−> s_2 ⟧
$\Longrightarrow$ <c_0;c_1, s> −c−> s_2

Natural Semantics: Sequential Composition

Semi ⟦ <c_0,s> −c−> s_1; <c_1,s_1> −c−> s_2 ⟧
$\Longrightarrow$ <c_0;c_1, s> −c−> s_2

Rationale of natural semantics:

- "jump" via c_0 from s to s_1, . . .

Natural Semantics: Sequential Composition

Semi ⟦ <c_0,s> −c−> s_1; <c_1,s_1> −c−> s_2 ⟧
$\Longrightarrow$ <c_0;c_1, s> −c−> s_2

Rationale of natural semantics:

- "jump" via c_0 from s to s_1, . . .
- . . . then 'jump" via c_1 from s_1 to s_2

Natural Semantics: Control Statements

IfTrue $\llbracket$ b s; <c_0,s> −c−> s_1 $\rrbracket$
$\Longrightarrow$ < IF b THEN c_0 ELSE c_1, s> −c−> s_1

IfFalse $\llbracket$ ¬b s; <c_1,s> −c−> s_1 $\rrbracket$
$\Longrightarrow$ < IF b THEN c_0 ELSE c_1, s> −c−> s_1

WhileFalse $\llbracket$¬b s$\rrbracket$
$\Longrightarrow$ <WHILE b DO c, s> −c−> s

WhileTrue $\llbracket$ b s; <c,s> −c−> s_1;
<WHILE b DO c,s_1> −c−> s_2 $\rrbracket$
$\Longrightarrow$ <WHILE b DO c, s> −c−> s_2

Note that for non-terminating programs no final state can be derived !

Transition Semantics

Transition semantics relates (inductively defined) "configurations".

Transition Semantics

Transition semantics relates (inductively defined) "configurations".

consts evalc1 :: ((com ×state) × (com ×state)) set
translations "cs_0 −1−> cs_1" ≡"(cs_0, cs_1) ∈ evalc1 "

Transition Semantics

Transition semantics relates (inductively defined) "configurations".

consts evalc1 :: ((com ×state) × (com ×state)) set
translations "cs_0 −1−> cs_1" ≡"(cs_0, cs_1) ∈ evalc1 "

We now start giving the actual inductive definition . . .

Transition Semantics: Assignment, Sequential Composition

inductive evalc1

intro

Assign (x:==a,s) −1−> (SKIP, s[x::=a s])

Semi1 (SKIP;c,s) −1−> (c,s)

Semi2 (c_0,s) −1−> (c_2,s_1)

$\Longrightarrow$ (c_0;c_1,s) −1−> (c_2;c_1,s_1)

Transition Semantics: Assignment, Sequential Composition

inductive evalc1

intro

Assign (x:==a,s) −1−> (SKIP, s[x::=a s])

Semi1 (SKIP;c,s) −1−> (c,s)

Semi2 (c_0,s) −1−> (c_2,s_1)

$\Longrightarrow$ (c_0;c_1,s) −1−> (c_2;c_1,s_1)

Rationale of Transition Semantics:

- the first component in a configuration represents a program counter . . .

Transition Semantics: Assignment, Sequential Composition

```
inductive evalc1
  intro
  Assign   (x:==a,s)  -1-> (SKIP, s[x::=a s])
  Semi1    (SKIP;c,s) -1-> (c,s)
  Semi2    (c_0,s)    -1-> (c_2,s_1)
           ==>  (c_0;c_1,s) -1-> (c_2;c_1,s_1)
```

Rationale of Transition Semantics:

- the first component in a configuration represents a program counter . . .
- transition semantics is close to an abstract machine.

Transition Semantics: Control Statements

IfTrue

$$b\ s \Longrightarrow (\texttt{IF}\ \ b\ \texttt{THEN}\ c_1\ \texttt{ELSE}\ c_2, s) \ -1-> (c_1, s)$$

IfFalse

$$\neg b\ s \Longrightarrow (\texttt{IF}\ \ b\ \texttt{THEN}\ c_1\ \texttt{ELSE}\ c_2, s) \ -1-> (c_2, s)$$

WhileFalse

$$\neg b\ s \Longrightarrow (\texttt{WHILE}\ \ b\ \texttt{DO}\ c, s) \ -1-> (\texttt{SKIP}, s)$$

WhileTrue

$$b\ s \Longrightarrow (\texttt{WHILE}\ \ b\ \texttt{DO}\ c, s) \ -1-> (c; \texttt{WHILE}\ b\ \texttt{DO}\ c, s)$$

Transition Semantics: Control Statements

IfTrue

$$b\ s \Longrightarrow (\texttt{IF}\ \ b\ \texttt{THEN}\ c_1\ \texttt{ELSE}\ c_2, s) \ -1-> (c_1, s)$$

IfFalse

$$\neg b\ s \Longrightarrow (\texttt{IF}\ \ b\ \texttt{THEN}\ c_1\ \texttt{ELSE}\ c_2, s) \ -1-> (c_2, s)$$

WhileFalse

$$\neg b\ s \Longrightarrow (\texttt{WHILE}\ \ b\ \texttt{DO}\ c, s) \ -1-> (\texttt{SKIP}, s)$$

WhileTrue

$$b\ s \Longrightarrow (\texttt{WHILE}\ \ b\ \texttt{DO}\ c, s) \ -1-> (c; \texttt{WHILE}\ b\ \texttt{DO}\ c, s)$$

A non-terminating loop always leads to successor configurations . . .

Generalized Step Relations

- n-step semantics:

Generalized Step Relations

- n-step semantics:

  "cs_0 −n−> cs_1" == "(cs_0,cs_1) ∈evalc1^n"

Generalized Step Relations

- n-step semantics:

  "cs_0 −n−> cs_1" == "(cs_0,cs_1) ∈evalc1^n"

- multistep-semantics:

Generalized Step Relations

- n-step semantics:

  "cs_0 −n−> cs_1" == "(cs_0,cs_1) ∈evalc1^n"

- multistep-semantics:

  "cs_0 −∗−> (c_1,s_1)" ≡"(cs_0,c_1 , s_1) ∈ evalc1^∗"

Equivalence

Natural semantics vs. transition semantics.

Theorem 10 (`evalc1_eq_evalc`):

$$(\mathtt{c},\ \mathtt{s})\ -*->\ (\mathtt{SKIP},\ \mathtt{t}) = (<\mathtt{c},\mathtt{s}>\ -\mathtt{c}->\ \mathtt{t})$$

Proof: by induction over com.

Denotational Semantics

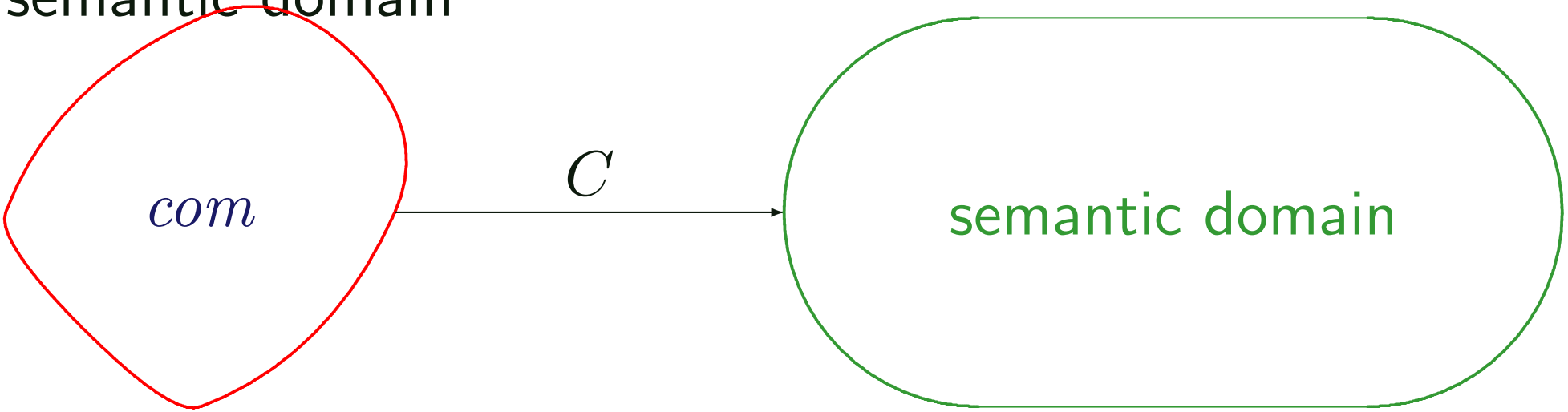
Idea:

Denotational Semantics

Idea: Explain recursion as fixpoint construction on the semantic domain

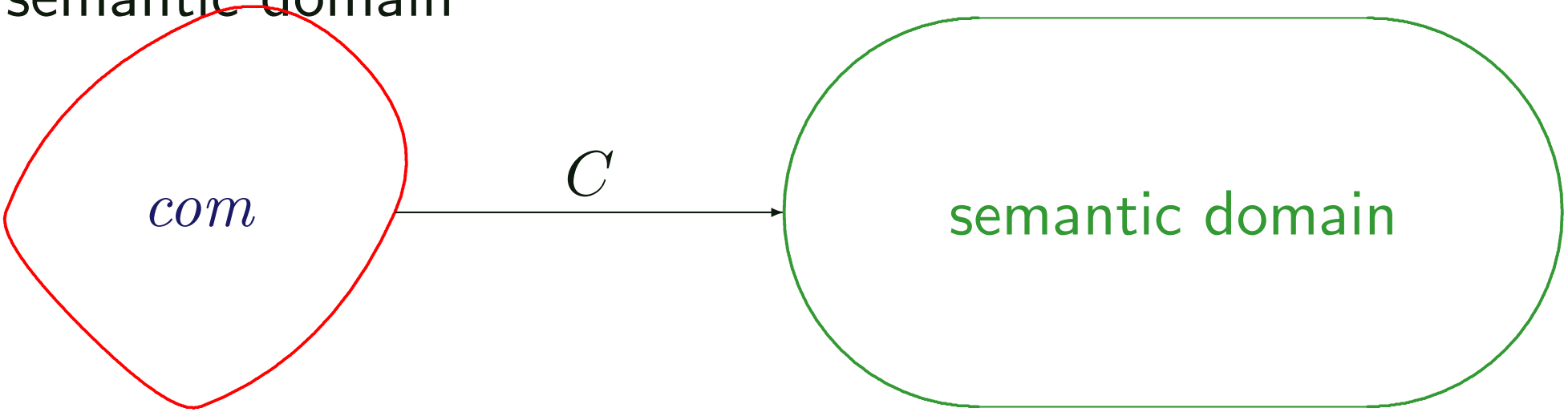
Denotational Semantics

Idea: Explain recursion as fixpoint construction on the semantic domain



Denotational Semantics

Idea: Explain recursion as fixpoint construction on the semantic domain

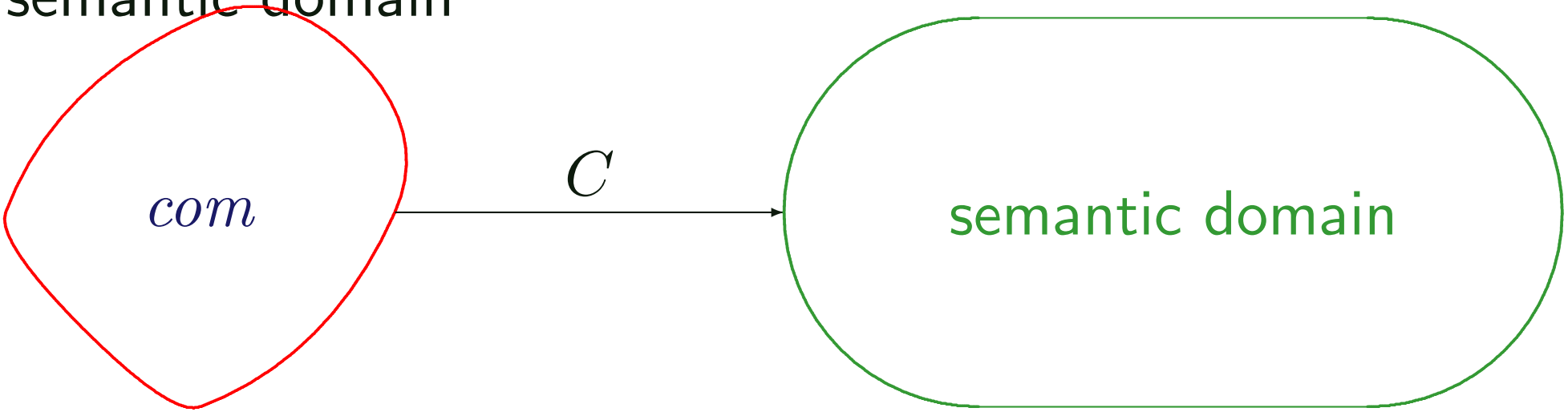


An (imperative) semantic domain is a state relation:

$$\text{com_den} = (\text{state} \times \text{state})\ \text{set}$$

Denotational Semantics

Idea: Explain recursion as fixpoint construction on the semantic domain



An (imperative) semantic domain is a state relation:

$$\text{com_den} = (\text{state} \times \text{state})\ \text{set}$$

Semantic function:

consts C :: com ⇒ com_den

The Inductive Definition

The semantics C is defined inductively:

The Inductive Definition

The semantics C is defined inductively:

primrec

```
C_skip     C(SKIP) = Id
C_assign   C(x :== a) = {(s,t). t = s[x::=a(s)]}
C_comp     C(c_0 ; c_1) = C(c_1) O C(c_0)
C_if       C( IF b THEN c_1 ELSE c_2) =
             {(s,t). (s,t) ∈C(c_1) ∧b(s)} ∪
             {(s,t). (s,t) ∈C(c_2) ∧¬b(s)}"
C_while    C( WHILE b DO c) = lfp (Gamma b (C c))"
```

The Inductive Definition

The semantics C is defined inductively:

primrec

```
C_skip     C(SKIP) = Id
C_assign   C(x :== a) = {(s,t). t = s[x::=a(s)]}
C_comp     C(c_0 ; c_1) = C(c_1) O C(c_0)
C_if       C( IF b THEN c_1 ELSE c_2) =
             {(s,t). (s,t) ∈C(c_1) ∧b(s)} ∪
             {(s,t). (s,t) ∈C(c_2) ∧¬b(s)}"
C_while    C( WHILE b DO c) = lfp (Gamma b (C c))"
```

where

```
Gamma b cd ≡(λphi.{(s,t). (s,t) ∈ (phi O cd) ∧b(s)} ∪
                  {(s,t). s=t ∧¬b(s)})
```

Equivalence of Programs

The following is an equivalence relating program fragments.

Equivalence of Programs

The following is an equivalence relating program fragments.

Theorem 11 (`C_While_If`):

```
C ( WHILE b DO c) =
C ( IF b THEN c; WHILE b DO c ELSE SKIP)
```

Equivalence of Programs

The following is an equivalence relating program fragments.

Theorem 11 (`C_While_If`):

```
C ( WHILE b DO c) =
C ( IF b THEN c; WHILE b DO c ELSE SKIP)
```

Such results justify program transformations.

Equivalence of Semantics

It turns out that denotational and natural semantics are equivalent in the following sense:

Equivalence of Semantics

It turns out that denotational and natural semantics are equivalent in the following sense:

Theorem 12 (denotational is natural):

$((s,\ t) \in C\ c) = (<c,s> -c-> t)$

Equivalence of Semantics

It turns out that denotational and natural semantics are equivalent in the following sense:

Theorem 12 (denotational is natural):

$((s,\ t) \in C\ c) = (<c,s> -c-> t)$

Still, if we want to prove properties of states to hold, we need different proof techniques. An answer to this need is axiomatic semantics or Hoare Logics.

Axiomatic Semantics

Idea:

Axiomatic Semantics

Idea:we relate "legal states" before and after a program execution.

Axiomatic Semantics

Idea:we relate "legal states" before and after a program execution. A set of legal states is an "assertion":

Axiomatic Semantics

Idea:we relate "legal states" before and after a program execution. A set of legal states is an "assertion":

types assn $=$ state $\Rightarrow$ bool

Hoare Logics

The key concept of a Hoare Logics is a Hoare Triple

consts hoare :: (assn × com × assn) set

translations "|– {P}c{Q}" ≡"(P,c,Q) ∈hoare"

Hoare Logics

The key concept of a Hoare Logics is a Hoare Triple

consts hoare :: (assn × com × assn) set

translations "|– {P}c{Q}" ≡"(P,c,Q) ∈hoare"

A triple has the intuitive meaning: if P holds for some state s and c terminates and reaches some state s' then Q must hold for s'.

Hoare Logics

The key concept of a Hoare Logics is a Hoare Triple

consts hoare :: (assn × com × assn) set

translations "|– {P}c{Q}" ≡"(P,c,Q) ∈hoare"

A triple has the intuitive meaning: if P holds for some state s and c terminates and reaches some state s' then Q must hold for s'.

The "logic" itself is an inductive definition: . . .

Hoare Logics: SKIP

inductive hoare

intro

skip "|– {P} SKIP {P}"

. . .

Hoare Logics: Assignment

ass "|– {λs. P(s[x::=(a s)])} x:==a {P}"

Hoare Logics: Assignment

ass "|– {λs. P(s[x::=(a s)])} x:==a {P}"

This may be counter-intuitive

Hoare Logics: Assignment

ass "|– {λs. P(s[x::=(a s)])} x:==a {P}"

This may be counter-intuitive but but consider the example $a \equiv \lambda s.1$ and $P \equiv \lambda s.\ s\,x = 1$

Hoare Logics: Assignment

ass "|– {λs. P(s[x::=(a s)])} x:==a {P}"

This may be counter-intuitive but but consider the example $a \equiv \lambda s.1$ and $P \equiv \lambda s.\ s\,x = 1$

$$\{\lambda s.(\lambda s.s\,x = 1)(s[x ::= 1])\}x :== \lambda s.1\{\lambda s.s\,x = 1\} \longrightarrow_\beta$$
$$\{\lambda s.(s[x ::= 1])x = 1\}x :== \lambda s.1\{\lambda s.s\,x = 1\} \longrightarrow_\beta$$
$$\{\lambda s.True\}x :== \lambda s.1\{\lambda s.s\,x = 1\}$$

Hoare Logics: Sequence,IF

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 $\{Q\}$"

Hoare Logics: Sequence,IF

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 $\{Q\}$"

The rule for `IF` represents, as expected, the case split.

Hoare Logics: WHILE

While

$$[\![\ |\!- \{\lambda s.\ P\ s \wedge b\ s\}\ c\ \{P\}\ ]\!]$$
$$\Longrightarrow |\!- \{P\}\ \texttt{WHILE}\ b\ \texttt{DO}\ c\ \{\lambda s.\ P\ s \wedge \neg b\ s\}"$$

Hoare Logics: `WHILE`

While

$$[\![\ |\!- \{\lambda s.\ P\ s \wedge b\ s\}\ c\ \{P\}\ ]\!]$$
$$\Longrightarrow |\!- \{P\}\ \texttt{WHILE}\ b\ \texttt{DO}\ c\ \{\lambda s.\ P\ s \wedge \neg b\ s\}"$$

If `WHILE` terminates, then the contrary of the condition must hold. . .

Hoare Logics: Consequence Rule

conseq $\llbracket\ \forall s.\ P'\ s \longrightarrow P\ s;$
$\vdash \{P\}c\{Q\};$
$\forall s.\ Q\ s \longrightarrow Q'\ s\ \rrbracket$
$\Longrightarrow \vdash \{P'\}c\{Q'\}$

Hoare Logics: Consequence Rule

$$
\begin{array}{ll}
\text{conseq} & [\![\ \forall s.\ P'\ s \longrightarrow P\ s; \\
& \quad \vdash \{P\}c\{Q\}; \\
& \quad \forall s.\ Q\ s \longrightarrow Q'\ s\ ]\!] \\
& \Longrightarrow \vdash \{P'\}c\{Q'\}
\end{array}
$$

One can always strengthen the pre-condition or weaken the post-condition.

Hoare Logics at a Glance

inductive hoare

intro

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 $[\![\forall s.P'\ s \longrightarrow P\ s; \vdash \{P\}c\{Q\};$

$$\forall s.\ Q\ s \longrightarrow Q'\ s] \Longrightarrow\ |-\ \{P'\}c\{Q'$$

Validity Relation

We define a validity relation:

$$\models \{P\}c\{Q\} \equiv \forall s.\ \forall t.\ (s,t) \in C(c) \longrightarrow P\ s \longrightarrow Q\ t''$$

Validity Relation

We define a validity relation:

$$\models \{P\}c\{Q\} \equiv \forall s.\ \forall t.\ (s,t) \in C(c) \longrightarrow P\ s \longrightarrow Q\ t"$$

Validity represents our intuition of what Hoare triples mean: whenever the program c can make a transition from s to t (wrt. to the underlying operational/denotational semantics), and whenever P holds for s, Q must hold for t.

Relating Hoare and Denotational Semantics

Theorem 13 (Hoare soundness):

$$\vdash \{P\}\, c\, \{Q\} \Longrightarrow \models \{P\}\, c\, \{Q\}$$

Theorem 14 (Hoare relative completeness):

$$\models \{P\}\, c\, \{Q\} \Longrightarrow \vdash \{P\}\, c\, \{Q\}$$

Why relative?

So the Hoare relation is in fact compatible with the denotational semantics of IMP.

Example Program

```
tm   :== λs. 1;
sum  :== λs. 1;
i    :== λs. 0;
 WHILE  λs. (s sum) <= (s a) DO
   (i    :== λs. (s i) + 1;
    tm   :== λs. (s tm) + 2;
    sum  :== λs. (s tm) + (s sum))
```

What does this program do?

Example Program

```
tm   :== λs. 1;
sum  :== λs. 1;
i    :== λs. 0;
 WHILE  λs. (s sum) <= (s a) DO
   (i    :== λs. (s i) + 1;
    tm   :== λs. (s tm) + 2;
    sum  :== λs. (s tm) + (s sum))
```

What does this program do?

Try $a = 1$, $a = 2$, . . . , and look at i!

Square Root

Answer: The program computes the square root. Informally:

$$Pre \equiv \text{"}True\text{"}$$

Square Root

Answer: The program computes the square root. Informally:

$$\begin{aligned} Pre &\equiv \text{"}True\text{"} \\ Post &\equiv \text{"}i^2 \leq a < (i+1)^2\text{"} \end{aligned}$$

Square Root

Answer: The program computes the square root. Informally:

$$\begin{aligned} Pre &\equiv \text{"}True\text{"} \\ Post &\equiv \text{"}i^2 \leq a < (i+1)^2\text{"} \end{aligned}$$

Formally

$$Pre \equiv \lambda s.\ True$$

Square Root

Answer: The program computes the square root. Informally:

$$\begin{aligned} Pre &\equiv \text{"}True\text{"} \\ Post &\equiv \text{"}i^2 \leq a < (i+1)^2\text{"} \end{aligned}$$

Formally

$$\begin{aligned} Pre &\equiv \lambda s.\ \ True \\ Post &\equiv \lambda s.\ \ (s\ i) * (s\ i) \leq (s\ a) \wedge \\ &\qquad\qquad s\ a < (s\ i + 1) * (s\ i + 1) \end{aligned}$$

Proving $\{Pre\} \ldots \{Post\}$

When using the Hoare-calculus directly:

- we apply the rules following the syntax
- we can apply the conseq-rule initially (interfacing Pre and $Post$
- we can apply the conseq-rule when entering a `WHILE` (interfacing the invariant)

Proving $\{Pre\} \ldots \{Post\}$

When using the Hoare-calculus directly:

- we apply the rules following the syntax
- we can apply the conseq-rule initially (interfacing Pre and $Post$
- we can apply the conseq-rule when entering a `WHILE` (interfacing the invariant)

Abbreviation: $ExC \equiv \lambda s.Inv\ s \wedge \neg s\ sum \leq s\ a$ ("exit condition").

Proving $\{Pre\} \ldots \{Post\}$

When using the Hoare-calculus directly:

- we apply the rules following the syntax
- we can apply the conseq-rule initially (interfacing Pre and $Post$
- we can apply the conseq-rule when entering a `WHILE` (interfacing the invariant)

Abbreviation: $ExC \equiv \lambda s.Inv\ s \wedge \neg s\ sum \leq s\ a$ ("exit condition"). We will develop the proof and from its structure "guess" the Invariant.

Proof

$$\{Pre\}\boxed{tm\ldots}\{Post\}$$

This is what we want to prove.

Proof

$$\frac{\boxed{} \quad \{ \qquad\qquad \} \boxed{tm\ldots} \{ExC\} \qquad\qquad \boxed{\mathcal{I}_2}}{\{Pre\}\boxed{tm\ldots}\{Post\}}\ conseq$$

Nothing happens after the loop, so intuition says that ExC must imply $Post$.

Proof

$$
\begin{array}{lll}
 & \{PW\}\boxed{WH\ldots}\{ExC\} & \\
 & \hline & semi \\
\{ & \}\boxed{i\ldots}\{ExC\} & \\
 & \hline & semi \\
\{ & \}\boxed{sum\ldots}\{ExC\} & \\
 & \hline & semi \\
\{ & \}\boxed{tm\ldots}\{ExC\} & \boxed{\mathcal{I}_2} \\
 & \hline & conseq \\
 & \{Pre\}\boxed{tm\ldots}\{Post\} &
\end{array}
$$

Apply $semi$ three times. PW ("pre while") is just a sensible choice of name: we don't know yet what it is.

Proof

$$\dfrac{\dfrac{\dfrac{\dfrac{\boxed{\mathcal{A}_3} \quad \{PW\}\boxed{WH\ldots}\{ExC\}}{\boxed{} \quad \{\quad\}\boxed{i\ldots}\{ExC\}}\,semi}{\boxed{} \quad \{\quad\}\boxed{sum\ldots}\{ExC\}}\,semi}{\boxed{} \quad \{\quad\}\boxed{tm\ldots}\{ExC\}}\,semi \qquad \boxed{\mathcal{I}_2}}{\{Pre\}\boxed{tm\ldots}\{Post\}}\,conseq$$

This application of ass will allow us to reconstruct the precondition in the line just below.

Proof

$$
\dfrac{\boxed{} \qquad \dfrac{\boxed{} \qquad \dfrac{\boxed{\mathcal{A}_2} \qquad \dfrac{\boxed{\mathcal{A}_3} \qquad \{PW\}\boxed{WH\ldots}\{ExC\}}{\{\lambda s.PW(s["i"])\}\boxed{i\ldots}\{ExC\}}\,semi}{\{\qquad\}\boxed{sum\ldots}\{ExC\}}\,semi}{\{\qquad\}\boxed{tm\ldots}\{ExC\}}\,semi \qquad \boxed{\mathcal{I}_2}}{\{Pre\}\boxed{tm\ldots}\{Post\}}\,conseq
$$

And likewise $\boxed{\mathcal{A}_2}$.

Proof

$$\dfrac{\boxed{} \qquad \dfrac{\boxed{\mathcal{A}_1} \qquad \dfrac{\boxed{\mathcal{A}_2} \qquad \dfrac{\boxed{\mathcal{A}_3} \qquad \{PW\}\boxed{WH\ldots}\{ExC\}}{\{\lambda s.PW(s["i"])\}\boxed{i\ldots}\{ExC\}}\,semi}{\{\lambda s.PW(s["i,sum"])\}\boxed{sum\ldots}\{ExC\}}\,semi}{\{\qquad\qquad\}\boxed{tm\ldots}\{ExC\}}\,semi \qquad \boxed{\mathcal{I}_2}}{\{Pre\}\boxed{tm\ldots}\{Post\}}\,conseq$$

And likewise $\boxed{\mathcal{A}_1}$.

Proof

$$
\begin{array}{l}
\boxed{\mathcal{I}_1} \quad \dfrac{\dfrac{\dfrac{\boxed{\mathcal{A}_3} \quad \{PW\}\boxed{WH\ldots}\{ExC\}}{\{\lambda s.PW(s["i"])\}\boxed{i\ldots}\{ExC\}}\,semi \quad \boxed{\mathcal{A}_2}}{\{\lambda s.PW(s["i,sum"])\}\boxed{sum\ldots}\{ExC\}}\,semi \quad \boxed{\mathcal{A}_1}}{\{\lambda s.PW(s["i,sum,tm"])\}\boxed{tm\ldots}\{ExC\}}\,semi \quad \boxed{\mathcal{I}_2}}{\{Pre\}\boxed{tm\ldots}\{Post\}}\,conseq
\end{array}
$$

We now know (by the form of $conseq$) what $\boxed{\mathcal{I}_1}$ is.

Proof

$$\boxed{\mathcal{I}_3}\ \{Inv\}\boxed{WH\ldots}\{ExC\}\ \boxed{\mathcal{I}_4}\ \ conseq$$

$$\boxed{\mathcal{A}_3}\ \{PW\}\boxed{WH\ldots}\{ExC\}\ \ semi$$

$$\boxed{\mathcal{A}_2}\ \{\lambda s.PW(s["i"])\}\boxed{i\ldots}\{ExC\}\ \ semi$$

$$\boxed{\mathcal{A}_1}\ \{\lambda s.PW(s["i,sum"])\}\boxed{sum\ldots}\{ExC\}\ \ semi$$

$$\boxed{\mathcal{I}_1}\ \{\lambda s.PW(s["i,sum,tm"])\}\boxed{tm\ldots}\{ExC\}\ \boxed{\mathcal{I}_2}\ \ conseq$$

$$\{Pre\}\boxed{tm\ldots}\{Post\}$$

Intuition says that PW must imply Inv.

Of course, we are not ready yet. . .

Completing the Proof

$\boxed{\mathcal{A}_1}$, $\boxed{\mathcal{A}_2}$ and $\boxed{\mathcal{A}_3}$ are complete, and $\boxed{\mathcal{I}_4}$ is trivial.

Completing the Proof

$\boxed{\mathcal{A}_1}$, $\boxed{\mathcal{A}_2}$ and $\boxed{\mathcal{A}_3}$ are complete, and $\boxed{\mathcal{I}_4}$ is trivial.

$\boxed{\mathcal{I}_1}$, $\boxed{\mathcal{I}_2}$, $\boxed{\mathcal{I}_3}$, and $\{Inv\}\boxed{WH\ldots}\{ExC\}$ remain to be shown.

Completing the Proof

$\boxed{\mathcal{A}_1}$, $\boxed{\mathcal{A}_2}$ and $\boxed{\mathcal{A}_3}$ are complete, and $\boxed{\mathcal{I}_4}$ is trivial.

$\boxed{\mathcal{I}_1}$, $\boxed{\mathcal{I}_2}$, $\boxed{\mathcal{I}_3}$, and $\{Inv\}\boxed{WH\ldots}\{ExC\}$ remain to be shown.

This also involves the question how the metavariables must be instantiated.

What is PW?

The metavariable PW ("precondition of $WHILE$") must fulfill (to show $\boxed{\mathcal{I}_1}$)

$$\forall s. \mathit{Pre}\ s \rightarrow PW(s[i ::= 0][sum ::= 1][tm ::= 1])$$

What is PW?

The metavariable PW ("precondition of $WHILE$") must fulfill (to show $\boxed{\mathcal{I}_1}$)

$$\forall s. Pre\ s \rightarrow PW(s[i ::= 0][sum ::= 1][tm ::= 1])$$

Solution (recall that $Pre \equiv \lambda s. True$):

$$PW = \lambda s. s\, i = 0 \wedge s\, sum = 1 \wedge s\, tm = 1$$

What is Inv?

Continuing our proof tree construction:

$$\frac{\{\lambda s.Inv\ s \wedge s\ sum \leq s\ a\}\ \boxed{\text{"body"}}\ \{Inv\}}{\{Inv\}\ \boxed{WH\ \ldots}\ \{ExC\}}\ \textit{While}$$

What is Inv?

Continuing our proof tree construction:

$$\dfrac{\dfrac{\{\lambda s.Inv\ s \wedge s\ sum \leq s\ a\} i :== \lambda s.s\ i + 1 \{P'\} \quad \{P'\} tm :== \lambda s.s\ tm + 2 \{P''\} \quad \{P''\} sum :== \lambda s.s\ tm + s\ sum \{Inv\}}{\{\lambda s.Inv\ s \wedge s\ sum \leq s\ a\} \boxed{\text{"body"}} \{Inv\}}\ semi^2}{\{Inv\} \boxed{WH \ldots} \{ExC\}}\ While$$

Just blindly applying $semi$ twice gives three formulas to be proven using ass, one for each assignment in the loop.

What is Inv?

Continuing our proof tree construction:

$$\dfrac{\dfrac{\{\lambda s.Inv\ s \wedge s\ sum \leq s\ a\} i :== \lambda s.s\ i + 1 \{P'\} \quad \{P'\} tm :== \lambda s.s\ tm + 2 \{P''\} \quad \{P''\} sum :== \lambda s.s\ tm + s\ sum \{Inv\}}{\{\lambda s.Inv\ s \wedge s\ sum \leq s\ a\} \boxed{\text{"body"}} \{Inv\}}\ semi^2}{\{Inv\} \boxed{WH \ldots} \{ExC\}}\ While$$

Just blindly applying $semi$ twice gives three formulas to be proven using ass, one for each assignment in the loop.

Now what are P' and P''? Have a look at rule ass first!

Calculating P' and P'' (by Rule ass)

$$P'' = \lambda s.Inv(s[sum ::= s\ tm + s\ sum])$$

Calculating P' and P'' (by Rule ass)

$$P'' = \lambda s.Inv(s[sum ::= s\ tm + s\ sum])$$

$$P' = \lambda s'.P''(s'[tm ::= s'\ tm + 2]) \qquad \text{(rule } ass\text{)}$$

Calculating P' and P'' (by Rule ass)

$$P'' = \lambda s.Inv(s[sum ::= s\ tm + s\ sum])$$

$$\begin{aligned} P' &= \lambda s'.P''(s'[tm ::= s'\ tm + 2]) \qquad \text{(rule } ass\text{)} \\ &= \lambda s'.(\lambda s.Inv(s[sum ::= s\ tm + s\ sum])) \\ &\qquad\qquad (s'[tm ::= s'\ tm + 2]) \end{aligned}$$

Calculating P' and P'' (by Rule ass)

$$P'' = \lambda s.Inv(s[sum ::= s\ tm + s\ sum])$$

$$\begin{aligned}
P' &= \lambda s'.P''(s'[tm ::= s'\ tm + 2]) \qquad \text{(rule } ass\text{)} \\
&= \lambda s'.(\lambda s.Inv(s[sum ::= s\ tm + s\ sum])) \\
&\qquad\qquad (s'[tm ::= s'\ tm + 2]) \\
&= \lambda s'.Inv((s'[tm ::= s'\ tm + 2]) \\
&\qquad\qquad [sum ::= (s'[tm ::= s'\ tm + 2])\ tm + \\
&\qquad\qquad\qquad (s'[tm ::= s'\ tm + 2])\ sum])
\end{aligned}$$

Calculating P' and P'' (by Rule ass)

$$P'' = \lambda s.Inv(s[sum ::= s\ tm + s\ sum])$$

$$\begin{aligned}
P' &= \lambda s'.P''(s'[tm ::= s'\ tm + 2]) \qquad \text{(rule } ass\text{)} \\
&= \lambda s'.(\lambda s.Inv(s[sum ::= s\ tm + s\ sum])) \\
&\qquad\qquad (s'[tm ::= s'\ tm + 2]) \\
&= \lambda s'.Inv((s'[tm ::= s'\ tm + 2]) \\
&\qquad\qquad [sum ::= (s'[tm ::= s'\ tm + 2])\ tm + \\
&\qquad\qquad\qquad (s'[tm ::= s'\ tm + 2])\ sum]) \\
&= \lambda s'.Inv(s'[tm ::= s'\ tm + 2] \\
&\qquad\qquad [sum ::= s'\ tm + 2 + s'\ sum]).
\end{aligned}$$

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 P for $\lambda s.Inv\ s \wedge s\ sum \leq s\ a$. Recall $P' =$

$$\lambda s.Inv(s[tm ::= s\ tm + 2][sum ::= s\ tm + 2 + s\ sum]).$$

Applying ass to $i :== \lambda s.s\, i + 1$

Now treat $i :== \lambda s.s\, i + 1$ in the same way. Temporarily, let's write P for $\lambda s.Inv\ s \wedge s\ sum \leq s\ a$. Recall $P' =$

$$\lambda s.Inv(s[tm ::= s\ tm + 2][sum ::= s\ tm + 2 + s\ sum]).$$

$P = \lambda s'.P'(s'[i ::= s'\ i + 1])$ (by rule ass)

Applying ass to $i :== \lambda s.s\, i + 1$

Now treat $i :== \lambda s.s\, i + 1$ in the same way. Temporarily, let's write P for $\lambda s.Inv\ s \wedge s\ sum \leq s\ a$. Recall $P' =$

$$\lambda s.Inv(s[tm ::= s\ tm + 2][sum ::= s\ tm + 2 + s\ sum]).$$

$$\begin{aligned} P &= \lambda s'.P'(s'[i ::= s'\ i + 1]) \qquad \text{(by rule } ass\text{)} \\ &= \lambda s'.(\lambda s.Inv(s[tm ::= s\ tm + 2][sum ::= s\ tm + 2 + s\ sum]))\\ &\qquad (s'[i ::= s'\ i + 1]) \end{aligned}$$

Applying ass to $i :== \lambda s.s\ i + 1$

Now treat $i :== \lambda s.s\ i + 1$ in the same way. Temporarily, let's write P for $\lambda s.Inv\ s \wedge s\ sum \leq s\ a$. Recall $P' =$

$$\lambda s.Inv(s[tm ::= s\ tm + 2][sum ::= s\ tm + 2 + s\ sum]).$$

$$\begin{aligned}
&P = \lambda s'.P'(s'[i ::= s'\ i + 1]) \qquad \text{(by rule } ass\text{)}\\
&= \lambda s'.(\lambda s.Inv(s[tm ::= s\ tm + 2][sum ::= s\ tm + 2 + s\ sum]))\\
&\qquad\qquad (s'[i ::= s'\ i + 1])\\
&= \lambda s'.Inv((s'[i ::= s'\ i + 1])\\
&\qquad [tm ::= (s'[i ::= s'\ i + 1])\ tm + 2]\\
&\qquad [sum ::= (s'[i ::= s'\ i + 1])\ tm + 2 + (s'[i ::= s'\ i + 1])\ sum]))
\end{aligned}$$

Applying ass to $i :== \lambda s.s\ i + 1$

Now treat $i :== \lambda s.s\ i + 1$ in the same way. Temporarily, let's write P for $\lambda s.Inv\ s \wedge s\ sum \leq s\ a$. Recall $P' =$

$$\lambda s.Inv(s[tm ::= s\ tm + 2][sum ::= s\ tm + 2 + s\ sum]).$$

$$\begin{aligned}
P &= \lambda s'.P'(s'[i ::= s'\ i + 1]) \qquad \text{(by rule } ass\text{)} \\
&= \lambda s'.(\lambda s.Inv(s[tm ::= s\ tm + 2][sum ::= s\ tm + 2 + s\ sum]))\\
&\qquad\qquad (s'[i ::= s'\ i + 1]) \\
&= \lambda s'.Inv((s'[i ::= s'\ i + 1]) \\
&\qquad [tm ::= (s'[i ::= s'\ i + 1])\ tm + 2] \\
&\qquad [sum ::= (s'[i ::= s'\ i + 1])\ tm + 2 + (s'[i ::= s'\ i + 1])\ sum])) \\
&= \lambda s.Inv(s[i ::= s\ i + 1][tm ::= s\ tm + 2][sum ::= s\ tm + 2 + s\ sum]).
\end{aligned}$$

Applying ass to $i :== \lambda s.s\ i + 1$

$$\lambda s.Inv\ s \wedge s\ sum \leq s\ a$$

$$= \lambda s.Inv(s[i ::= s\ i + 1][tm ::= s\ tm + 2][sum ::= s\ tm + 2 + s\ sum]).$$

So Inv must solve this equation.

Inv Must Fulfill the Equation

Inv must fulfill the equation

$$\begin{aligned}&\lambda s.Inv\ s \wedge s\ sum \leq s\ a = \\ &\lambda s.Inv(s[i ::= s\ i + 1][tm ::= s\ tm + 2] \\ &\qquad\qquad [sum ::= s\ tm + 2 + s\ sum])\end{aligned}$$

Inv Must Fulfill the Equation

Inv must fulfill the equation

$$\begin{aligned}&\forall s.Inv\ s \wedge s\ sum \leq s\ a = \\ &\forall s.Inv(s[i ::= s\ i + 1][tm ::= s\ tm + 2] \\ &\qquad\qquad [sum ::= s\ tm + 2 + s\ sum])\end{aligned}$$

We can replace λ by $\forall$ due to extensionality.

Inv Must Fulfill the Equation

Inv must fulfill the equation

$$\begin{aligned}&\forall s.Inv\ s \wedge s\ sum \leq s\ a = \\ &\forall s.Inv(s[i ::= s\ i + 1][tm ::= s\ tm + 2] \\ &\qquad\qquad [sum ::= s\ tm + 2 + s\ sum])\end{aligned}$$

We can replace λ by $\forall$ due to extensionality.

Guessing the right Inv is obviously difficult! Informally

$$Inv \ \equiv\ "(i+1)^2 = sum\ \wedge\ tm = (2 * i) + 1\ \wedge\ i^2 \leq a"$$

Checking that Inv Fulfills equation

$$s\ sum \leq s\ a \quad \wedge \tag{6}$$

$$(s\ i + 1)^2 = (s\ sum) \quad \wedge \tag{7}$$

$$s\ tm = (2 * (s\ i)) + 1 \quad \wedge \tag{8}$$

$$(s\ i)^2 \leq (s\ a) \quad \wedge \tag{9}$$

$$\text{(recall: } = \text{ means } \leftrightarrow\text{)} \quad = \tag{10}$$

$$((s\ i + 1) + 1)^2 = (s\ sum) + (s\ tm) + 2 \quad \wedge \tag{11}$$

$$(s\ tm + 2) = (2 * (s\ i + 1)) + 1 \quad \wedge \tag{12}$$

$$(s\ i + 1)^2 \leq (s\ a) \tag{13}$$

Proof Sketch

First show the "$\longrightarrow$"-direction:

(8) $\longrightarrow$ (12) and (6) $\wedge$ (7) $\longrightarrow$ (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 "$\leftarrow$"-direction:

(12) $\rightarrow$ (8) and (13) $\rightarrow$ (9) by simple arithmetic. (7) is shown as follows:

$$
\begin{aligned}
(s\,i+1)^2 &= ((s\,i+1)+1)^2 - 2*(s\,i+1) - 1 \\
&\overset{(11)}{=} (s\,sum) + (s\,tm) + 2 - 2*(s\,i+1) - 1 \\
&\overset{(12)}{=} (s\,sum) + 2*(s\,i+1) + 1 \\
&\qquad\qquad -2*(s\,i+1) - 1 \\
&= s\,sum
\end{aligned}
$$

Finally, (7) $\wedge$ (13) $\rightarrow$ (6).

Proof Sketch (Cont.)

Now show the "$\leftarrow$"-direction:

(12) $\rightarrow$ (8) and (13) $\rightarrow$ (9) by simple arithmetic. (7) is shown as follows:

$$
\begin{aligned}
(s\,i+1)^2 &= ((s\,i+1)+1)^2 - 2*(s\,i+1) - 1 \\
&\overset{(11)}{=} (s\,sum) + (s\,tm) + 2 - 2*(s\,i+1) - 1 \\
&\overset{(12)}{=} (s\,sum) + 2*(s\,i+1) + 1 \\
&\qquad\qquad -2*(s\,i+1) - 1 \\
&= s\,sum
\end{aligned}
$$

Finally, (7) $\wedge$ (13) $\rightarrow$ (6). So Inv is indeed an invariant!

The $WHILE$ Loop: Remarks

We have shown

$$(\text{“enter condition”} \wedge \text{“invar. at entry”}) \leftrightarrow \text{“invar. at exit”}$$

The $WHILE$ Loop: Remarks

We have shown

$$(\text{“enter condition”} \wedge \text{“invar. at entry”}) \leftrightarrow \text{“invar. at exit”}$$

One would definitely expect $\rightarrow$, but $\leftarrow$ is remarkable!

The $WHILE$ Loop: Remarks

We have shown

(“enter condition” $\wedge$ “invar. at entry”)$\leftrightarrow$“invar. at exit”

One would definitely expect $\rightarrow$, but $\leftarrow$ is remarkable!

We can show this because our invariant is so strong: for showing $\rightarrow$, the weaker invariant (7) $\wedge$ (8), i.e.

$$”(i+1)^2 = sum \ \wedge \ tm = (2 * i) + 1$$

would do (check it!).

The $WHILE$ Loop: Remarks

We have shown

$$(\text{“enter condition”} \wedge \text{“invar. at entry”}) \leftrightarrow \text{“invar. at exit”}$$

One would definitely expect $\rightarrow$, but $\leftarrow$ is remarkable!

We can show this because our invariant is so strong: for showing $\rightarrow$, the weaker invariant (7) $\wedge$ (8), i.e.

$$\text{”}(i+1)^2 = sum \ \wedge \ tm = (2 * i) + 1$$

would do (check it!).

But the extra condition $i^2 \leq a$ is needed for showing $Post$, which states what the program actually computes.

Taking Care of $Post$

We have shown $\boxed{\mathcal{I}_1}$ and $\{Inv\}\boxed{WH\ldots}\{ExC\}$. Now continue with $\boxed{\mathcal{I}_2}$.

Does $Post\ s$ follow from $Inv\ s \wedge \neg s\ sum \leq s\ a$?

Taking Care of $Post$

We have shown $\boxed{\mathcal{I}_1}$ and $\{Inv\}\boxed{WH\ldots}\{ExC\}$. Now continue with $\boxed{\mathcal{I}_2}$.

Does $Post\ s$ follow from $Inv\ s \wedge \neg s\ sum \leq s\ a$?

Yes!

$(s\ i)^2 \leq (s\ a)$	follows from (9)
$(s\ a) < (s\ i + 1)^2$	follows from $\neg s\ sum \leq (s\ a)$ and (7).

The Final Missing Part

$\boxed{\mathcal{I}_3}$ remains to be shown, i.e.

$$\forall s. PW\ s \rightarrow Inv\ s$$

or, expanding the solutions for PW and Inv

$$\begin{aligned}\forall s.\quad & s\,i = 0 \wedge s\,sum = 1 \wedge s\,tm = 1 \rightarrow \\ & (s\,i + 1)^2 = s\,sum \wedge \\ & s\,tm = (2 * (s\,i)) + 1 \wedge \\ & (s\,i)^2 \leq (s\,a)\end{aligned}$$

This is easy to check.

An Alternative for Tackling the Loop Part

Recall that our loop invariant was "too strong". An alternative:

$$\frac{\{\lambda s.Inv\ s \wedge s\ sum \leq s\ a\}\boxed{\text{"body"}}\{Inv\}}{\{Inv\}\boxed{WH\ldots}\{ExC\}}\ While$$

An Alternative for Tackling the Loop Part

Recall that our loop invariant was "too strong". An alternative:

$$\dfrac{\dfrac{\forall s.(Inv\ s \wedge s\ sum \leq s\ a) \rightarrow Inv'\ s \qquad \{Inv'\}\boxed{\text{"body"}}\{Inv\}}{\{\lambda s.Inv\ s \wedge s\ sum \leq s\ a\}\boxed{\text{"body"}}\{Inv\}}\ conseq}{\{Inv\}\boxed{WH\ldots}\{ExC\}}\ While$$

An Alternative for Tackling the Loop Part

Recall that our loop invariant was "too strong". An alternative:

$$\dfrac{\dfrac{\forall s.(Inv\ s \wedge s\ sum \leq s\ a) \rightarrow Inv'\ s \qquad \dfrac{\{Inv'\}i :== \lambda s.s\ i + 1\{P'\} \quad \{P'\}tm :== \lambda s.s\ tm + 2\{P''\} \quad \{P''\}sum :== \lambda s.s\ tm + s\ sum\{Inv\}}{\{Inv'\}\boxed{\text{"body"}}\{Inv\}}\ semi^2}{\{\lambda s.Inv\ s \wedge s\ sum \leq s\ a\}\boxed{\text{"body"}}\{Inv\}}\ conseq}{\{Inv\}\boxed{WH\ldots}\{ExC\}}\ While$$

Alternative (Cont.)

Applying ass as before gives

$$Inv' = \lambda s.Inv(s[i ::= s\,i + 1][tm ::= s\,tm + 2] \\ [sum ::= s\,tm + 2 + s\,sum])$$

We are left with the proof obligation

$$\forall s.(Inv\ s \wedge s\,sum \leq s\,a) \rightarrow \\ Inv(s[i ::= s\,i + 1][tm ::= s\,tm + 2] \\ [sum ::= s\,tm + 2 + s\,sum])$$

Automating Hoare Proofs

In the example, we have verified a program computing the square root.

But this was tedious, and parts of the task can be automated.

Weakest Preconditions

Observation: the Hoare relation is deterministic to a certain extent.

Weakest Preconditions

Observation: the Hoare relation is deterministic to a certain extent.

Idea: we use this fact for the generation of weakest preconditions.

Weakest preconditions are:

constdefs wp :: com ⇒ assn ⇒ assn
"wp c Q ≡ (λs. ∀t. (s, t) ∈ C(c) ⟶ Q t)"

So $wp\ c\ Q$ returns the set of states containing all states s such that if t is reached from s via c, then the post-condition Q holds for t. Computable?

Weakest Preconditions

Observation: the Hoare relation is deterministic to a certain extent.

Idea: we use this fact for the generation of weakest preconditions.

Weakest preconditions are:

constdefs wp :: com ⇒ assn ⇒ assn
"wp c Q ≡ (λs. ∀t. (s, t) ∈ C(c) ⟶ Q t)"

So $wp\ c\ Q$ returns the set of states containing all states s such that if t is reached from s via c, then the post-condition Q holds for t. Computable? Not obvious.

Equivalence Proofs

Main results of the wp-generator are:

wp_SKIP: $\text{wp SKIP Q} = \text{Q}$

wp_Ass: $\text{wp(x :== a) Q} = (\lambda\text{s. Q (s[x::=a s])})$

wp_Semi: $\text{wp(c; d) Q} = \text{wp c (wp d Q)}$

wp_If: $\text{wp (IF b THEN c ELSE d) Q} =$
$(\lambda\text{s. (b s} \longrightarrow \text{wp c Q s)} \wedge (\neg\text{b s} \longrightarrow \text{wp d Q s}))$

wp_While_True: $\text{b s} \Longrightarrow \text{wp (WHILE b DO c) Q s} =$
$\text{wp (c; WHILE b DO c) Q s}$

wp_While_False: $\neg\text{b s} \Longrightarrow \text{wp (WHILE b DO c) Q s} = \text{Q s}$

wp_While_if: $\text{wp (WHILE b DO c) Q s} =$
$\text{(if b s then wp(c; WHILE b DO c) Q s else Q s)}$

Computing Weakest Preconditions

Except for termination problem due to While, weakest precondition wp can be computed.

This fact can be used for further proof support by verification condition generation.

Idea: for all statements, the exact wp is computed, except for the creative step at While, where the assertion INV provided by the user for the invariant is taken.

Computing Weakest Preconditions

Except for termination problem due to While, weakest precondition wp can be computed.

This fact can be used for further proof support by verification condition generation.

Idea: for all statements, the exact wp is computed, except for the creative step at While, where the assertion INV provided by the user for the invariant is taken. An additional function vc ("verification condition") establishes necessary conditions that INV is indeed an invariant.

Annotated IMP Programs

We enrich the syntax by loop-invariants:

Annotated IMP Programs

We enrich the syntax by loop-invariants:

```
datatype acom =
    Askip
  | Aass loc aexp
  | Asemi acom acom
  | Aif bexp acom acom
  | Awhile bexp assn acom
```

Computing a "Approximative" Weakest Precondition

We define a function that computes an "approximative" wp:

Computing a "Approximative" Weakest Precondition

We define a function that computes an "approximative" wp:

primrec

$$\begin{aligned}
&\text{awp Askip Q} = \text{Q}\\
&\text{awp (Aass x a) Q} = (\lambda\text{s. Q(s[x::=a s])})\\
&\text{awp (Asemi c d) Q} = \text{awp c (awp d Q)}\\
&\text{awp (Aif b c d) Q} = (\lambda\text{s. (b s}\longrightarrow\text{awp c Q s)} \wedge (\neg\text{b s}\longrightarrow\text{awp d Q s}))\\
&\text{awp (Awhile b Inv c) Q} = \text{Inv}
\end{aligned}$$

Note that awp is not necessarily a wp; this depends if Inv is indeed an invariant.

Verification Condition Generation

Inv is an Invariant if verification conditionvc holds:

primrec

$$
\begin{aligned}
&\mathrm{vc}\ \mathrm{Askip}\ \mathrm{Q} = (\lambda \mathrm{s}.\ \mathrm{True}) \\
&\mathrm{vc}\ (\mathrm{Aass}\ \mathrm{x}\ \mathrm{a})\ \mathrm{Q} = (\lambda \mathrm{s}.\ \mathrm{True}) \\
&\mathrm{vc}\ (\mathrm{Asemi}\ \mathrm{c}\ \mathrm{d})\ \mathrm{Q} = (\lambda \mathrm{s}.\ \mathrm{vc}\ \mathrm{c}\ (\mathrm{awp}\ \mathrm{d}\ \mathrm{Q})\ \mathrm{s} \wedge \mathrm{vc}\ \mathrm{d}\ \mathrm{Q}\ \mathrm{s}) \\
&\mathrm{vc}\ (\mathrm{Aif}\ \mathrm{b}\ \mathrm{c}\ \mathrm{d})\ \mathrm{Q} = (\lambda \mathrm{s}.\ \mathrm{vc}\ \mathrm{c}\ \mathrm{Q}\ \mathrm{s} \wedge \mathrm{vc}\ \mathrm{d}\ \mathrm{Q}\ \mathrm{s}) \\
&\mathrm{vc}\ (\mathrm{Awhile}\ \mathrm{b}\ \mathrm{Inv}\ \mathrm{c})\ \mathrm{Q} = (\lambda \mathrm{s}.\ (\mathrm{Inv}\ \mathrm{s} \wedge \neg \mathrm{b}\ \mathrm{s} \longrightarrow \mathrm{Q}\ \mathrm{s}) \wedge \\
&\qquad\qquad (\mathrm{Inv}\ \mathrm{s} \wedge \mathrm{b}\ \mathrm{s} \longrightarrow \mathrm{awp}\ \mathrm{c}\ \mathrm{Inv}\ \mathrm{s}) \wedge \\
&\qquad\qquad \mathrm{vc}\ \mathrm{c}\ \mathrm{Inv}\ \mathrm{s})
\end{aligned}
$$

Results on vc (1)

The following facts on vc and awp makes this concept powerful:

Results on vc (1)

The following facts on vc and awp makes this concept powerful:

Theorem 15 (Soundness (`vc_sound`)):

$(\forall s.\ \mathrm{vc}\ \mathrm{ac}\ Q\ s) \Longrightarrow \vdash \{\mathrm{awp}\ \mathrm{ac}\ Q\}\ \mathrm{astrip}\ \mathrm{ac}\ \{Q\}$

Results on vc (1)

The following facts on vc and awp makes this concept powerful:

Theorem 15 (Soundness (`vc_sound`)):

$(\forall \text{s. vc ac Q s}) \Longrightarrow \vdash \{\text{awp ac Q}\}\ \text{astrip ac}\ \{\text{Q}\}$

vc generated from the annotated program holds, then there exists a Hoare-proof for the program without annotations showing that the postcondition follows from its weakest (annotated) precondition.

Results on vc (2)

Moreover, we have:

Results on vc (2)

Moreover, we have:

Theorem 16 (Completeness (`vc_complete`)):

$$\vdash \{P\}\ c\ \{Q\} \Longrightarrow \exists\, ac.\ \ \text{astrip}\ \ ac = c \wedge (\forall s.\ \text{vc}\ ac\ Q\ s) \wedge (\forall s.\ P\ s \longrightarrow \text{awp}\ ac\ Q\ s)$$

Results on vc (2)

Moreover, we have:

Theorem 16 (Completeness (`vc_complete`)):

$$\vdash \{P\}\ c\ \{Q\} \Longrightarrow \exists\, ac.\ \ \text{astrip}\ \ ac = c \wedge (\forall s.\ \text{vc}\ ac\ Q\ s) \wedge (\forall s.\ P\ s \longrightarrow \text{awp}\ ac\ Q\ s)$$

If a Hoare-proof exists, there must exist an annotated program, for which vc generates a true formula and whose precondition P implies the its weakest precondition.

Quintessence: vc abstracts Hoare-proofs away from (imperative) program verification. . .

Summary

- IMP closely follows the standard textbook [Win96].

Summary

- IMP closely follows the standard textbook [Win96].
- Isabelle/HOL is a powerful framework for embedding imperative languages.

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.

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.
- Even verification condition generators can be proven sound and complete within HOL.

More Detailed Explanations

Equivalence Proofs

Summarizing, we have the following equivalence results:

- natural vs. transition semantics
- denotational vs. natural semantics

Locations

We realize program variables via pointers (locations). The type of pointers is an abstract datatype.

Defining of values by nat is just a simplification.

A state is a function taking a location to a value, i.e. intuitively, each program variable corresponds to a location, each access to a program variable is an application of state to the location of this variable.

The Intuition of Natural Semantics

The idea of the natural semantics is that a program relates two states, the "input state" and the "output state", provided that it terminates.

This is similar to denotational at first sight, but the treatment of "recursive" constructs such as the `WHILE` is different: denotational semantics reduces these to fixpoint operators, natural semantics to the (meta)-question, if a derivation for the transition is possible or not.

The Intuition of Transition Semantics

Unlike the natural semantics, the transition semantics records the single steps of the computation. A configuration is a pair consisting of a program and a state, and one step reaches a new program and a new state.

The intuition behind the program-component is "the program to be executed"; it is manipulated in a stack-like manner during the evaluation of the rules.

An in-depth investigation of the rules reveals, that there are only finitely many different programs-components in all configuration traces; these correspond to "positions" in a program. (Note that assignments have a "position before" and a "position after" execution). Thus, this component can be seen as a program counter (PC).

Understanding Gamma

We discuss the approximation relation Gamma in more detail:

"Gamma b cd $\equiv$(λphi.{(s,t). (s,t) : (phi O cd) $\wedge$ b(s)} $\cup$
{(s,t). s=t $\wedge\neg$b(s)})"

Note that in the definition of `WHILE`, the second argument cd to Gamma is used for the meaning of the body of the loop.

Thus, the underlying principle is similar to the one used when defining the transitive closure by lfp.

Let

$$S_0 = \{(s,t) \mid s = t \wedge \neg b(s)\}$$

be the initial apprimation (the subset of the identity relation, for which the condition b is false.

Then we can iterate from S_0 via composition cd^n arbitrarily many

transitions through the body of the loop. The lfp represents the limit of this approximation process.

Note that for b $= \lambda$x. true S_0 is empty, therefore Gamma b cd is empty and, consequently, the denotational semantics C will yield the empty relation in such cases.

A Table of Values

a is not modified anywhere. Therefore a can be seen as input of the program.

A Table of Values

a is not modified anywhere. Therefore a can be seen 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:

A Table of Values

a is not modified anywhere. Therefore a can be seen as input of the program.

i counts the number of times the loop is entered, i.e. the final value of i is the number of times the loop was entered. This number depends on a. The following table shows that final values of i, tm and sum depending on the value of a:

	i	tm	sum
$0 \leq a < 1$	0	1	1
$1 \leq a < 4$	1	3	4
$4 \leq a < 9$	2	5	9
$9 \leq a < 16$	3	7	16
$16 \leq a < 25$	4	9	25
$25 \leq a < 36$	5	11	36
$36 \leq a < 49$	6	13	49

sum takes the values of all squares successively, computed by the famous binomial formula:

$$(i+1)^2 = i^2 + 2i + 1$$

Since tm takes the value $2i+1$ for all i successively, it follows that

$sum + tm$ always gives the next value of sum.

$(s\ i)$, $(s\ a)$ etc.

Informally we talk about variables i, x etc. and say "x has value 5", for example. But formally, program variables are realized via locations, and when accessing a program variable, we get expressions of the form $s\ x$. That is, $s\ x$ is the value of variable x.

Nondeterminacy in the Hoare Calculus

The $conseq$ rule can always be applied. For all other commands, the choice for a hoare-rule is uniquely determined.

Relative Completeness

After Gödel, logicians tend to be nervous whenever a logic is claimed to be complete, in particular if arithmetic is involved as is the case for IMP.

Relative completeness means that for any valid Hoare triple in validity relation there will be a proof in hoare logic.

However, will we be able to derive in HOL that any semantically valid hoare triple is valid in the sense of validity relation ? What if we write in a precondition something like "if Goldbach's conjecture holds" or, worse, "if we can solve an arbitrary diophantine equation"?

The answer is no since HOL itself is incomplete wrt. standard models.

Program "Fragment"

This is the entire program, namely:

```
tm   :== λs. 1;
sum  :== λs. 1;
i    :== λs. 0;
 WHILE  λs.  s sum <= s a DO
   (i     :== λs. s i + 1;
    tm    :== λs. s tm + 2;
    sum   :== λs. s tm + s sum)
```

(return to main proof tree)

Program Fragment

This is the program fragment starting from sum :==, namely:

```
sum  :== λs. 1;
i    :== λs. 0;
 WHILE  λs.  s sum <= s a DO
   (i     :== λs. s i + 1;
    tm    :== λs. s tm + 2;
    sum   :== λs. s tm + s sum)
```

(return to main proof tree)

Program Fragment

This is the program fragment starting from i :==, namely:

```
i     :== λx. 0;
WHILE  λs.  s  sum <= s a DO
  (i     :== λs. s  i  + 1;
   tm  :== λs. s tm + 2;
   sum  :== λs. s tm + s sum)
```

(return to main proof tree)

Invariant

$$Inv \;\equiv\; \text{"}\lambda s.(s\,i+1)^2 = s\,sum \;\wedge\; s\,tm = (2 * s\,i) + 1 \;\wedge\; s\,i^2 \leq a\text{"}$$

Program Fragment

This is the program fragment starting from $WHILE$, namely:

```
WHILE λs. s sum <= s a DO
  (i   :== λs. s i + 1;
   tm  :== λs. s tm + 2;
   sum :== λs. s tm + s sum)
```

(return to main proof tree)

Program Fragment

This is the program fragment consisting of the loop body, namely:

$$\begin{aligned} \mathsf{i} &:== \lambda \mathsf{s}.\ \mathsf{s}\ \mathsf{i} + 1; \\ \mathsf{tm} &:== \lambda \mathsf{s}.\ \mathsf{s}\ \mathsf{tm} + 2; \\ \mathsf{sum} &:== \lambda \mathsf{s}.\ \mathsf{s}\ \mathsf{tm} + \mathsf{s}\ \mathsf{sum} \end{aligned}$$

(return to main proof tree)

A Missing Part

$\boxed{\mathcal{I}_1}$ is the formula

$$\forall s. Pre\ s \rightarrow PW(s["i, sum, tm"])$$

where Pre is defined above and PW is a metavariable ("precondition of $WHILE$").

(return to main proof tree)

A Missing Part

$\boxed{\mathcal{I}_2}$ is the formula

$$\forall s.ExC\ s \rightarrow Post\ s,$$

i.e.

$$\forall s.Inv\ s \wedge \neg sum\ s \leq s\ a \rightarrow Post\ s,$$

where $Post$ is defined above and Inv is a metavariable ("loop invariant").

(return to main proof tree)

A Missing Part

$\boxed{\mathcal{A}_1}$ is the proof tree

$$\frac{}{\{\lambda s.PW(s["i, sum, tm"])\}tm :== \lambda x.1\{\lambda s.PW(s["i, sum"])\}}\ ass$$

where PW is a metavariable ("precondition of $WHILE$").

(return to main proof tree)

A Missing Part

$\boxed{\mathcal{A}_2}$ is the proof tree

$$\frac{}{\{\lambda s.PW(s[i ::= 0][sum ::= 1])\}sum :== \lambda x.1\{\lambda s.PW(s[i ::= 0])\}}\ ass$$

where PW is a metavariable ("precondition of $WHILE$").

(return to main proof tree)

A Missing Part

$\boxed{\mathcal{A}_3}$ is the proof tree

$$\frac{}{\{\lambda s.PW(s[i ::= 0])\}i :== \lambda x.0\{PW\}}\ ass$$

where PW is a metavariable ("precondition of $WHILE$").

(return to main proof tree)

A Missing Part

$\boxed{\mathcal{I}_3}$ is the formula

$$\forall s.PW\ s \rightarrow Inv\ s$$

where PW is a metavariable ("precondition of $WHILE$") and Inv is a metavariable ("loop invariant").

(return to main proof tree)

A Missing Part

$\boxed{\mathcal{I}_4}$ is the formula

$$\forall s.PW\ s \rightarrow PW\ s$$

which is of course trivial to prove.

(return to main proof tree)

An Abbreviation for an Updated State

We use $s["i, sum, tm"]$ as abbreviation for

$$s[i ::= 0][sum ::= 1][tm ::= 1]$$

(return to main proof tree)

An Abbreviation for an Updated State

We use $s["i, sum"]$ as abbreviation for

$$s[i ::= 0][sum ::= 1]$$

(return to main proof tree)

An Abbreviation for an Updated State

We use $s["i"]$ as abbreviation for

$$s[i ::= 0]$$

(return to main proof tree)

What must Inv Be?

Recall 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. 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.

Higher-Order Logic Application: Denotational Semantics for Functional Languages

Burkhart Wolff

Global Outline (1)

- Foundations
  - Foundational Axioms, Methodology, Historical Background, Principia Structure
  - Fixpoints and Inductive Sets
  - Well-founded Orders and Recursors
  - Arithmetic, Data-Types

Global Outline (2)

- Embeddings
  - Foundations,
    Functional Languages and Denotational Semantics
  - Imperative Languages, Refinement Calculus
  - Z and Data-Refinement,
    CSP and Process-Refinement
  - Object-oriented Languages (Java-Light . . .)

Global Outline (2)

- Embeddings
  - Foundations,
    Functional Languages and Denotational Semantics
  - Imperative Languages, Refinement Calculus
  - Z and Data-Refinement,
    CSP and Process-Refinement
  - Object-oriented Languages (Java-Light . . .)

Motivation

- Current stage of our course:
  - we have a logical framework
    for computer science
  - with set theory,
    total function recursion theory
  - proof support for: inductive sets,
    datatypes, primitive recursion definition
  - rich library

Motivation

- Current stage of our course:
  - we have a logical framework
    for computer science
  - with set theory,
    total function recursion theory
  - proof support for: inductive sets,
    datatypes, primitive recursion definition
  - rich library

$\Rightarrow$ how can we apply this framework to specification and programming languages?

Representation Techniques for Semantics

Outline:

- Representing Languages in HOL

Outline:

- Representing Languages in HOL
  - shallow
  - deep

Outline:

- Representing Languages in HOL
  - shallow
  - deep
- Foundation for Functional Programming
  - sets and relations
  - cpo's

Outline:

- Representing Languages in HOL
  - shallow
  - deep
- Foundation for Functional Programming
  - sets and relations
  - cpo's
- Deriving Operational Semantics . . .

Question:

What is the *Meaning* of a "Language"?

Syntax and Semantics

- syntax: language = set of symbols

Syntax and Semantics

- syntax: language = set of symbols
- semantics:
  - set of denotations, the "semantic *domain*"
  - meaning function (or: *interpretation*) relating these two

Syntax and Semantics

A Language:

- set of "words" (strings)

  ⇒ concrete syntax

  - definition techniques:
    inductive sets of strings in HOL

Syntax and Semantics

A Language:

- set of "words" (strings)

  ⇒ concrete syntax

  ◦ definition techniques:
  inductive sets of strings in HOL

- set of "trees" (terms)

  ⇒ abstract syntax

  ◦ definition techniques:

  ▷ abstract data types

  ▷ constant definitions in HOL

Representation Techniques: An Example

Example: Regular Expressions

- The Language:
  - concrete syntax in a BNF-grammar:

$$
\begin{aligned}
\text{rex} &::= \text{char}\ \text{"}-\text{"}\ \text{char}\\
\text{rex} &::= \text{char}\\
\text{rex} &::= \text{"}.\text{"}\\
\text{rex} &::= \text{"("}\ \text{rex}\ \text{")"}\\
\text{rex} &::= \text{"["}\ \text{rex}\ \text{"]"}\\
\text{rex} &::= \text{rex}\ \text{"}*\text{"}\\
\text{rex} &::= \text{rex}\ \text{"|"}\ \text{rex}\\
\text{rex} &::= \text{rex}\ \text{rex}
\end{aligned}
$$

Example: Regular Expressions

- The Language:
  - concrete syntax: BNF-grammar as inductive definition (Version 1):

consts rex :: string set

inductive "rex"

intros

range	$[(x :: \text{char})]$ @ "–" @ $[(y :: \text{char})] \in \text{rex}$
char	$[(x :: \text{char})] \in \text{rex}$
dot	"." $\in$ rex
par	$r_1 \in \text{rex} \Longrightarrow$ "(" @ r_1 @ ")" $\in$ rex
bracket	$r_1 \in \text{rex} \Longrightarrow$ "[" @ r_1 @ "]" $\in$ rex
star	$r_1 \in \text{rex} \Longrightarrow r_1$ @ "$*$" $\in$ rex
alt	$[\![r_1 \in \text{rex};\ r_2 \in \text{rex}]\!] \Longrightarrow r_1$ @ "|" @ $r_2 \in$ rex
seq	$[\![r_1 \in \text{rex};\ r_2 \in \text{rex}]\!] \Longrightarrow r_1$ @ $r_2 \in$ rex"

Example: Regular Expressions

- The Language:
  - Well-known problems: grammars not deterministic, . . .

Example: Regular Expressions

- The Language:
  - Well-known problems: grammars not deterministic, . . .
  - therefore precedences, auxiliary non-terminals, . . .

$$
\begin{array}{lcl}
\text{rex} & ::= & \text{sx}\ [\text{"|"}\ \text{rx}] \\
\text{sx} & ::= & \text{tx}\ [\text{sx}] \\
\text{tx} & ::= & \text{ax}\ [\text{"*"} \mid \text{"+"} \mid \text{"?"}] \\
\text{ra} & ::= & \text{char}\ \text{"-"}\ \text{char} \\
\text{mx} & ::= & \text{ra}\ [\text{mx}] \\
\text{ax} & ::= & \text{char} \\
\text{ax} & ::= & \text{"."} \\
\text{ax} & ::= & \text{"("}\ \text{rex}\ \text{")"} \\
\text{ax} & ::= & \text{"["}\ \text{mx}\ \text{"]"}
\end{array}
$$

Example: Regular Expressions

- The Language:
  - (well-known) solution: *abstract* syntaxes implemented as data-type (Version 2):

Example: Regular Expressions

- The Language:
  - (well-known) solution: *abstract* syntaxes implemented as data-type (Version 2):

```
datatype rex =
range char char ("_-_")
  | char char ("(_)")
  | dot (".")
  | bracket rex ("[_]")
  | star rex ("_*")
  | alt rex rex ("_ |_")
  | seq rex rex ("_ _")
```

Example: Regular Expressions

- The Language:
  - (well-known) solution: *abstract* syntaxes implemented as data-type (Version 2):

```
datatype rex =
range char char ("_-_")
  | char char ("(_)")
  | dot (".")
  | bracket rex ("[_]")
  | star rex ("_*")
  | alt rex rex ("_ |_")
  | seq rex rex ("_ _")
```

- Note:
  - no "par"-variant necessary!
  - priorities ommited!

Example: Regular Expressions

- The Language:
  - (well-known) solution: *abstract* syntaxes implemented as signature (Version 3):

Example: Regular Expressions

- The Language:
  - (well-known) solution: *abstract* syntaxes implemented as signature (Version 3):

```
type rex
  consts
  range   :: [char, char] ⇒ rex ("_–_")
  char    :: char ⇒ rex ("_")
  dot     :: rex (".")
  bracket :: rex ⇒ rex ("[ _ ]")
  star    :: rex ⇒ rex ("_*")
  alt     :: [rex, rex] ⇒ rex ("_ | _")
  seq     :: [rex, rex] ⇒ rex ("_ _")
```

Example: Regular Expressions

- The Language:
  - Input into Isabelle:
    can be identical for all three versions,
    but highly different in their internal representation!

$$\mathrm{a}(\mathrm{c} - \mathrm{d})*$$

(provided a, c and d are the usual character constants . . .)

Question:

How can we represent semantics?

Semantic Representation

- Deep Embeddings:
  - syntax as *explicit* datatype (e.g. Version 2)
  - interpretation as explicit function mapping each element of the language to a value

Semantic Representation

- Deep Embeddings:
  - syntax as *explicit* datatype (e.g. Version 2)
  - interpretation as explicit function mapping each element of the language to a value

- Shallow Embedding:
  - syntax implicit in *notation* for operators on the semantic domain (based on Version 3)

Semantic Representation: Example

- Deep Embeddings (based on Version 2):
  - semantic function L primitive recursive:

Semantic Representation: Example

- Deep Embeddings (based on Version 2):
  - semantic function L primitive recursive:

consts L :: rex ⇒ string set

primrec L

L(range x y) = {[a] | a. x ≤ a ∧ a ≤ y}
L(char x) = {[x]}
L(dot) = {[x] | x. True}
L(bracket r) = {[]} ∪ L(r)
L(star r) = lfp(λX.{[]} ∪ {x@y| x,y. x∈ L(r) ∧ y ∈ X})
L(alt r_1 r_2) = L(r_1) ∪ L(r_2)
L(seq r_1 r_2) = {x @ y | x,y. x ∈ L(r_1) ∧ y ∈ L(r_2)}

where {f a | a. P a} ≡ f ‘{a. P a}

Representation

- Deep Embeddings: Question

Why does Version 1 does not work here for use with primitive recursion?

Semantic Representation: Example

- Shallow Embeddings (based on Version 3)
  - Operators are directly interpreted in domain:

type rex = string set

defs

range_def	range x y $\equiv \{[a] \mid a.\ x \leq a \wedge a \leq y\}$
char_def	char x $\equiv \{[x]\}$
dot_def	dot $\equiv \{[x] \mid x.\ \text{True}\}$
bracket_def	bracket r $\equiv \{[]\} \cup$ r
star_def	star r $\equiv$ lfp $(\lambda X.\{[]\} \cup \{x@y \mid x,y.\ x \in r \wedge y \in X\})$
alt_def	alt r_1 r_2 $\equiv$ r_1 $\cup$ r_2
seq_def	seq r_1 r_2 $\equiv \{x\ @\ y \mid x,y.\ x \in r_1 \wedge y \in r_2\}$

where $\{f\ a \mid a.\ P\ a\} \equiv f`\{a.\ P\ a\}$

Semantic Representation: Example

- Shallow Embeddings (based on Version 3):
  - Can we induce over shallow embeddings?

Semantic Representation: Example

- Shallow Embeddings (based on Version 3):
  - Can we induce over shallow embeddings?
  - Yes, if we can give the domain an inductive structure, e.g.:

Semantic Representation: Example

- Shallow Embeddings (based on Version 3):

consts REX :: string set set

inductive "REX"

intros

range	$\text{range } x\ y \in \text{REX}$
char	$\text{char } x \in \text{REX}$
dot	$\text{dot} \in \text{REX}$
bracket	$r_1 \in \text{REX} \Longrightarrow \{\text{bracket } x \mid x.\ x \in r_1\} \in \text{REX}$
star	$r_1 \in \text{REX} \Longrightarrow \{\text{star } x \mid x.\ x \in r_1\} \in \text{REX}$
alt	$[\![r_1 \in \text{REX}; r_2 \in \text{REX}]\!] \Longrightarrow$ $\{\text{alt } x\ y \mid x,y.\ x \in r_1 \wedge y \in r_2\} \in \text{REX}$
seq	$[\![r_1 \in \text{REX}; r_2 \in \text{REX}]\!] \Longrightarrow$ $\{\text{seq } x\ y \mid x,y.\ x \in r_1 \wedge y \in r_2\} \in \text{REX}$

- from REX to rex we can go via type definition . . .

Semantic Representation: Example

- Shallow Embeddings (based on Version 3):
  - The main advantage: we inherit the binding structure of HOL!

Semantic Representation: Example

- Shallow Embeddings (based on Version 3):
  - The main advantage: we inherit the binding structure of HOL!
  - Example: We can add

$$\mu X.f\ X = lfp\ f$$

into the rex-language and have:

$$\mu X.a[X]a = \{a^{2n}.n\}!!!$$

Semantic Representation: Example

- Shallow Embeddings (based on Version 3):
  - Note: In

$$\mu X.a[X]a = \{a^{2n}.n\},$$

X is a HOL-Variable, a a constant, etc.

In a deep embedding, we would have to introduce a constant set (e.g. $X\ 1$, $X\ 2$, . . .) for variables and handle them in own substitution functions . . .

Semantic Representation: Example

- Shallow Embeddings (based on Version 3):
  - Note: In

$$\mu X.a[X]a = \{a^{2n}.n\},$$

X is a HOL-Variable, a a constant, etc.

In a deep embedding, we would have to introduce a constant set (e.g. $X\ 1$, $X\ 2$, . . .) for variables and handle them in own substitution functions . . .

and take care of name clashes and name captures ourselves !

Semantic Representation: Example

- Shallow Embeddings (based on Version 3):
  - The main advantage:
    we inherit the binding structure of HOL!
  - In deep embeddings, this requires extra

Semantic Representation: Example

- Shallow Embeddings (based on Version 3):
  - The main advantage:
    we inherit the binding structure of HOL!
  - In deep embeddings, this requires extra

    variable symbols,

Semantic Representation: Example

- Shallow Embeddings (based on Version 3):
  - The main advantage:
    we inherit the binding structure of HOL!
  - In deep embeddings, this requires extra

    variable symbols,

    substitution functions,

Semantic Representation: Example

- Shallow Embeddings (based on Version 3):
  - The main advantage:
    we inherit the binding structure of HOL!
  - In deep embeddings, this requires extra

    variable symbols,

    substitution functions,

    typing functions, . . .

Semantic Representation: Example

- Shallow Embeddings (based on Version 3):
  - Drawbacks:
    - ▷ binding structure may be too tight (blockstructured, λ-calculus oriented)

Semantic Representation: Example

- Shallow Embeddings (based on Version 3):
  - Drawbacks:
    - ▷ binding structure may be too tight (blockstructured, λ-calculus oriented)
    - ▷ typing may be too tight

Semantic Representation: Example

- Shallow Embeddings (based on Version 3):
  - Drawbacks:
    - ▷ binding structure may be too tight (blockstructured, λ-calculus oriented)
    - ▷ typing may be too tight
    - ▷ the semantic domain may not have an inductive structure (α,$\alpha \Rightarrow \beta$, $\alpha\ set$)

Semantic Representation: Example

- Shallow Embeddings (based on Version 3):
  - Drawbacks:
    - ▷ binding structure may be too tight (blockstructured, λ-calculus oriented)
    - ▷ typing may be too tight
    - ▷ the semantic domain may not have an inductive structure (α,$\alpha \Rightarrow \beta$, $\alpha\ set$)
    - ▷ unfortunately, this rules out some crucial meta-language proofs (e.g. completeness proofs)

Semantic Representation

- Further Examples:
  - Deep Embeddings:

Semantic Representation

- Further Examples:
  - Deep Embeddings:
    - Theory/Collection "Lambda" (Isabelle99)

Semantic Representation

- Further Examples:
  - Deep Embeddings:
    - Theory/Collection "Lambda" (Isabelle99)
    - Languages IMP, NanoJava, ProofPower-Z
  - Shallow Embeddings:

Semantic Representation

- Further Examples:
  - Deep Embeddings:
    - ▷ Theory/Collection "Lambda" (Isabelle99)
    - ▷ Languages IMP, NanoJava, ProofPower-Z
  - Shallow Embeddings:
    - ▷ HOL itself !!! (λ, $\exists$, @, . . .)

Semantic Representation

- Further Examples:
  - Deep Embeddings:
    - ▷ Theory/Collection "Lambda" (Isabelle99)
    - ▷ Languages IMP, NanoJava, ProofPower-Z
  - Shallow Embeddings:
    - ▷ HOL itself !!! (λ, $\exists$, @, . . .)
    - ▷ HOLCF ([MNOS99]; see Isabelle distribution)

Semantic Representation

- Further Examples:
  - Deep Embeddings:
    - ▷ Theory/Collection "Lambda" (Isabelle99)
    - ▷ Languages IMP, NanoJava, ProofPower-Z
  - Shallow Embeddings:
    - ▷ HOL itself !!! (λ, $\exists$, @, . . .)
    - ▷ HOLCF ([MNOS99]; see Isabelle distribution)
    - ▷ HOL-Z [BRW03a], HOL-CSP [TW97]

Semantic Representation

- Further Examples:
  - Deep Embeddings:
    - ▷ Theory/Collection "Lambda" (Isabelle99)
    - ▷ Languages IMP, NanoJava, ProofPower-Z
  - Shallow Embeddings:
    - ▷ HOL itself !!! (λ, $\exists$, @, . . .)
    - ▷ HOLCF ([MNOS99]; see Isabelle distribution)
    - ▷ HOL-Z [BRW03a], HOL-CSP [TW97]
    - ▷ MiniML (discussed in the following)

Semantic Representation

- Further Examples:
  - Deep Embeddings:
    - Theory/Collection "Lambda" (Isabelle99)
    - Languages IMP, NanoJava, ProofPower-Z
  - Shallow Embeddings:
    - HOL itself !!! (λ, $\exists$, @, . . .)
    - HOLCF ([MNOS99]; see Isabelle distribution)
    - HOL-Z [BRW03a], HOL-CSP [TW97]
    - MiniML (discussed in the following)
  - In-between: MicroJava [NOP00]

Scott's Approach to Denotational Semantics

- Overall aim: Having a structure with a fixpoint property:

$$\mu X.E(X) = E(\mu X.E(X))$$

- Topological Approach
  (denotational, Scott/Stratchey Approach):

  $(\sqsubseteq, C)$ is a "complete partial order" (CPO)
- Idea: We represent $(\sqsubseteq, C)$ as subclass of order!

Denotational Semantics

- Some basic definitions:

upper bound	$S <\mid x \equiv \forall\, y.\ y \in S \rightarrow y \leq x$
is_least_ub	$S <<\mid x \equiv S <\mid x \wedge (\forall\, u.\ S <\mid u \rightarrow x \leq u)$
lub	$\mathrm{lub}(S) \equiv \mathrm{THE}\ x.\ S <<\mid x$
directed	$\mathrm{directed}\ X \equiv (X \neq \{\}) \wedge$
	$(\forall\, a \in X,\ b \in X.\ \exists\, c \in X.\ a \leq c \wedge b \leq c)$
fix	$\mathbf{fix}\ f \equiv \mathrm{lub}(\mathrm{range}(\lambda\, i.\ \ \mathrm{iterate}\ \ i\ F\ \bot))$

Denotational Semantics

- Some basic definitions:

classes cpo0 $<$ order

consts $\bot$ :: 'a :: cpo0 *(*In cpo's there is a constant Bottom *)*

axclass

cpo $<$ cpo0

least "$\bot \leq$x" *(* ... which is least *)*

complete "(directed X) $\Longrightarrow$ ($\exists$ b. X $<<|$ b)

Denotational Semantics

- Some basic definitions:
  - continuity:

$$\text{cont } f \equiv \forall\, Y.\ Y \subseteq A \wedge \text{directed } Y \rightarrow f`Y <<| f(\text{lub } Y)$$

where $f ::' a :: \text{cpo} \Rightarrow 'b :: \text{cpo}$

Denotational Semantics

- Some basic definitions:
  - continuity:

    $$\text{cont } f \equiv \forall\, Y.\ Y \subseteq A \wedge \text{directed } Y \rightarrow f`Y <<| f(\text{lub } Y)$$

    where $f ::' a :: \text{cpo} \Rightarrow 'b :: \text{cpo}$

  - admissibility:

    $$\text{adm}(P) \equiv$$
    $$\forall\, Y.\ Y \subseteq A \wedge \text{directed } Y \rightarrow (\forall\, x \in Y.\ P\ x) \rightarrow P(\text{lub } Y)$$

    where $P ::' a :: \text{cpo} \Rightarrow \text{bool}$

Denotational Semantics

- Main Theorems:
  - recursion (Knaster-Tarski):

    $$\text{cont } f \Longrightarrow \textbf{fix } f = f\,(\textbf{fix } f)$$

    where $f :: 'a :: cpo \Rightarrow 'a :: cpo$

  - fixpoint induction:

    $$[\![\ \text{cont } f;\ \text{adm}(P);\ P(\bot);\ \forall x.\ P(x) \Longrightarrow P(f\ x)]\!] \Longrightarrow P(\textbf{fix } f)$$

Denotational Semantics

- Minor Theorems:

$(\forall\ x.\ \mathrm{cont}(f\ x)) \Longrightarrow \mathrm{cont}(\lambda\ x.\ \textbf{fix}\ (f\ x))$

$\mathrm{cont}(\mathrm{Pair}),\ \mathrm{cont}(\mathrm{fst}), \mathrm{cont}(\mathrm{snd})$

$\mathrm{cont}(\lambda\ x.\ c)$

$\mathrm{cont}(\lambda\ x.\ x)$

$\mathrm{cont}(f) \wedge \mathrm{cont}(g) \Longrightarrow \mathrm{cont}(f \circ g)$

$\mathrm{cont}(u) \wedge \mathrm{cont}(v) \Longrightarrow \mathrm{adm}(\lambda\ x.\ u\ x \leq v\ x)$

$\mathrm{cont}(u) \wedge \mathrm{cont}(v) \Longrightarrow \mathrm{adm}(\lambda\ x.\ u\ x \leq v\ x)$

$\mathrm{adm}\ P \wedge \mathrm{adm}\ Q \Longrightarrow \mathrm{adm}(\lambda\ x. P\ x \wedge Q\ x)$

$\mathrm{adm}\ P \wedge \mathrm{adm}\ Q \Longrightarrow \mathrm{adm}(\lambda\ x. P\ x \vee Q\ x)$

Denotational Semantics

- Constructions: The "flat cpo"

datatype 'a up = lift 'a | down
instance up :: (term) ord

constdefs
drop :: 'a :: cpo up ⇒ 'a
"drop x ≡ case x **of** lift v ⟶ v | down ⟶ ⊥"

le_up_def "x ≤ y ≡ case x **of**
lift v ⟶ (case y **of** lift u ⟶ v = u
| down ⟶ False)
| down ⟶ True"

Denotational Semantics

- Extremely Nifty Constructions: Products, Function Space.

instance

"$\times$" :: (ord, ord) ord

defs

le_pair_def "x $\leq$ y $\equiv$ ((fst x $\leq$ fst y) $\wedge$ (snd x $\leq$ snd y))"

arities

fun :: (term, order) order $(* \; fun = \longrightarrow *)$

defs

le_fun_def "f $\leq$ g $\equiv$ ($\forall$x. f x $\leq$ g x)

Denotational Semantics

- Extremely Nifty Consequences

Dom2 = Dom1 +

instance up :: (term) cpo (⊥_up_def, le_up_least , le_up_complete)

instance

"×" :: (cpo,cpo) cpo (⊥_pair_def, le_pair_least ,

instance

fun :: (term,cpo) cpo (le_fun_least , le_fun_complete)

⇒ CPO-checking is type checking!

Example: Shallow Embedding of MiniML

- Core Definitions of MiniML [Win96]:
  one exception (EXN), apply, closure, if_then_else, . . .

```
EXN_def      EXN ≡ ⊥
APPLY_def    F ^! x ≡ if x = ⊥ then ⊥
                      else if F = ⊥ then ⊥ else (drop F) x
LAM_def      Lam f ≡ lift f        (* LAM x. f x*)
REC_def      REC f ≡ fix f
IF _def        ( IF x THEN y ELSE z) ≡ (case x of
                 lift v ⇒ if v then y else z
               | down ⇒ ⊥ )
LET_def      LET s f ≡ f ^! s
```

Denotational Semantics

- More Definitions of MiniML [Win96]:
  Basic Operations, Global Declarations.

types Int = int up Bool = bool up

constdefs TIMES :: [Int, Int] ⇒ Int
"TIMES ≡ strictify (λx::nat. strictify (λy. lift (x∗y)))"
DIV :: [Int, Int] ⇒ Int
"DIV ≡ strictify (λx::nat. strictify (λy.
if y = lift 0 then ⊥ else lift (x∗y))))"

constdefs VAL :: "['a,'a] ⇒ bool"
"VAL f E ≡(f = E)"

$$\text{FUN} :: \text{"}[\text{'a} :: \text{cpo}, \text{'a} \Rightarrow \text{'a}] \Rightarrow \text{bool"}$$

$$\text{"FUN f F} \equiv (\text{f} = \text{REC(F)}) \wedge \text{cont F"}$$

Denotational Semantics

- Now we can derive the operational semantics of MiniML: evaluation relation, canonical forms [Win96]

```
constdefs eval :: ['a::cpo,'a] ⇒ bool
          "eval s t ≡(s = t)"
          cf :: ['a::cpo] ⇒ bool
          "cf t ≡(t~=⊥)"

syntax    "−A −> "::['a,'a}] ⇒ bool ( infixl 50)

translations
          "s −A −> t" ≡"eval s t"
```

Denotational Semantics

- Now we can derive the operational semantics of MiniML: strict beta-reduction, if_then_else with EXN (=⊥).

```
⟦cf c; cf c_2;
  t_1 -A -> (LAM x. t x);
  t_2 -A -> c_2; (t(c_2)) -A -> c ⟧
⟹ (t_1 ^! t_2) -A -> c

⟦t_1 -A -> EXN; t_2 -A -> c_2⟧
⟹ (t1 ^! t_2) -A -> EXN

⟦t_1 -A -> (LAM x. t x);
```

$$
\begin{array}{l}
\quad \mathsf{t_2}\ \ -\mathsf{A}\ -\!\!>\ \mathsf{EXN}\ \rrbracket \\
\Longrightarrow (\mathsf{t1}\ \ \hat{}!\ \ \mathsf{t_2})\ \ -\mathsf{A}\ -\!\!>\ \mathsf{EXN}
\end{array}
$$

$$
\begin{array}{l}
\llbracket \mathsf{cf}\ \ \mathsf{c_2};\ \ \mathsf{t_1}\ \ -\mathsf{A}\ -\!\!>\ \mathsf{TRUE}; \\
\quad \mathsf{t_2}\ \ -\mathsf{A}\ -\!\!>\ \mathsf{c_2}\rrbracket \\
\Longrightarrow (\ \mathtt{IF}\ \ \mathsf{t_1}\ \ \mathtt{THEN}\ \ \mathsf{t_2}\ \ \mathtt{ELSE}\ \ \mathsf{t_3})\ \ -\mathsf{A}\ -\!\!>\ \mathsf{c_2}
\end{array}
$$

$$
\begin{array}{l}
\llbracket \mathsf{cf}\ \ \mathsf{c_2};\ \ \mathsf{t_1}\ \ -\mathsf{A}\ -\!\!>\ \mathsf{FALSE};\ \mathsf{t_3}\ -\mathsf{A}\ -\!\!>\ \mathsf{c_2}\ \rrbracket \\
\Longrightarrow (\ \mathtt{IF}\ \ \mathsf{t_1}\ \ \mathtt{THEN}\ \ \mathsf{t_2}\ \ \mathtt{ELSE}\ \ \mathsf{t_3})\ \ -\mathsf{A}\ -\!\!>\ \mathsf{c_2}
\end{array}
$$

$$
\begin{array}{l}
\llbracket \mathsf{t_1}\ \ -\mathsf{A}\ -\!\!>\ \mathsf{EXN}\ \rrbracket \\
\Longrightarrow (\ \mathtt{IF}\ \ \mathsf{t_1}\ \ \mathtt{THEN}\ \ \mathsf{t_2}\ \ \mathtt{ELSE}\ \ \mathsf{t_3})\ \ -\mathsf{A}\ -\!\!>\ \mathsf{EXN}
\end{array}
$$

Denotational Semantics

- Now we can derive the operational semantics of MiniML: recursion, basic operations.

$$
\begin{array}{l}
[\![\mathsf{cont}(\lambda X.\ \ \mathsf{lift}\ \ (f\ \ X))\ ]\!] \Longrightarrow \\
\quad \mathsf{REC}(\lambda X.(\mathsf{LAM}\ x_1.\ f\ X\ x_1))\ -\mathsf{A}\ -> \\
\qquad (\mathsf{LAM}\ x_1.\ f\ (\mathsf{REC}(\lambda X.\ (\mathsf{LAM}\ x_1.\ f\ X\ x_1)))\ x_1)
\end{array}
$$

$$
\begin{array}{l}
[\![\mathsf{cf}\ \ c_1;\ \ \mathsf{cf}\ \ c_2;\ \ t_1\ \ -\mathsf{A}\ -> c_1;\ t_2\ -\mathsf{A}\ -> c_2\ ]\!] \Longrightarrow \\
\quad (\mathsf{TIMES}\ t_1\ t_2)\ -\mathsf{A}\ -> \mathsf{lift}(c_1 * c_2)
\end{array}
$$

Summary

- Isabelle/HOL is a powerful framework for embedding programming languages
  - via deep embeddings
  - via shallow embeddings
  - denotational semantics can be used as definitional basis for operational semantics !!!

Summary

- Isabelle/HOL is a powerful framework for embedding programming languages
  - via deep embeddings
  - via shallow embeddings
  - denotational semantics can be used as definitional basis for operational semantics !!!

- First Priciple for language embeddings: KEEP IT AS SHALLOW AS POSSIBLE !!!

Higher-Order Logic Applications: Refinements

Burkhart Wolff

Overview

In previous weeks, we saw various embeddedings in HOL:

Overview

In previous weeks, we saw various embeddedings in HOL:

- Imperative languages

Overview

In previous weeks, we saw various embeddedings in HOL:

- Imperative languages
- Functional languages

Overview

In previous weeks, we saw various embeddedings in HOL:

- Imperative languages
- Functional languages
- Fragments of Specification Lanuages (HOL, Z)

Overview

In previous weeks, we saw various embeddedings in HOL:

- Imperative languages
- Functional languages
- Fragments of Specification Lanuages (HOL, Z)

Can we apply these theories to development methods such as

Refinement ?

Overview

In previous weeks, we saw various embeddedings in HOL:

- Imperative languages
- Functional languages
- Fragments of Specification Lanuages (HOL, Z)

Can we apply these theories to development methods such as

Refinement ?

Can we apply HOL to prove the relations between functions, operations, processes, architectures?

Rough Overview

Various Refinement Methods are described in the literature:

- Observational/Behavioural Equivalence

Rough Overview

Various Refinement Methods are described in the literature:

- Observational/Behavioural Equivalence
- Forget/Restrict/Identify-Method

Rough Overview

Various Refinement Methods are described in the literature:

- Observational/Behavioural Equivalence
- Forget/Restrict/Identify-Method
- Operation Refinement, Data Refinement [Spi92]

Rough Overview

Various Refinement Methods are described in the literature:

- Observational/Behavioural Equivalence
- Forget/Restrict/Identify-Method
- Operation Refinement, Data Refinement [Spi92]
- Refinement Calculus

Rough Overview

Various Refinement Methods are described in the literature:

- Observational/Behavioural Equivalence
- Forget/Restrict/Identify-Method
- Operation Refinement, Data Refinement [Spi92]
- Refinement Calculus
- Process Refinement (CSP [A.W97])

Rough Overview

Various Refinement Methods are described in the literature:

- Observational/Behavioural Equivalence
- Forget/Restrict/Identify-Method
- Operation Refinement, Data Refinement [Spi92]
- Refinement Calculus
- Process Refinement (CSP [A.W97])
- Machine Refinement (B-Method [Abr96])
- . . .

(thousands of articles and many books on the subject. Arbitrary selection by the author).

Common Formal Method Classification

One destinguishes:

- Data-Oriented Modelling Techniques:

Common Formal Method Classification

One destinguishes:

- Data-Oriented Modelling Techniques:

  one system step involving
  complex transformation of
  input, output and state data,

Common Formal Method Classification

One destinguishes:

- Data-Oriented Modelling Techniques:

  one system step involving
  complex transformation of
  input, output and state data,

- Behavioral Modelling:

Common Formal Method Classification

One destinguishes:

- Data-Oriented Modelling Techniques:

  one system step involving
  complex transformation of
  input, output and state data,

- Behavioral Modelling:

  sequences of system steps considering
  the evolution of input, output and states.

Common Formal Method Classification

One destinguishes:

- Data-Oriented Modelling Techniques:

Common Formal Method Classification

One destinguishes:

- Data-Oriented Modelling Techniques:

  data refinement(Z, KIV, B),
  algebraic specification techniques
  (Behavioural Equivalence),
  Hoare-like calculi (Morgan, Back/Wright)

Common Formal Method Classification

One destinguishes:

- Data-Oriented Modelling Techniques:

  data refinement(Z, KIV, B),
  algebraic specification techniques
  (Behavioural Equivalence),
  Hoare-like calculi (Morgan, Back/Wright)

- Behavioral Modelling:

Common Formal Method Classification

One destinguishes:

- Data-Oriented Modelling Techniques:

  data refinement(Z, KIV, B),
  algebraic specification techniques
  (Behavioural Equivalence),
  Hoare-like calculi (Morgan, Back/Wright)

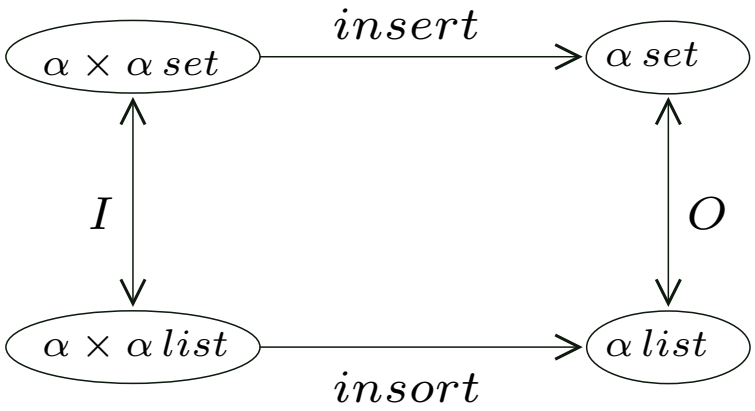
- Behavioral Modelling:

  process algebras (CSP,CCS,. . .),
  temporal logics

Data Refinement for a Function

A simple example for refining a function:

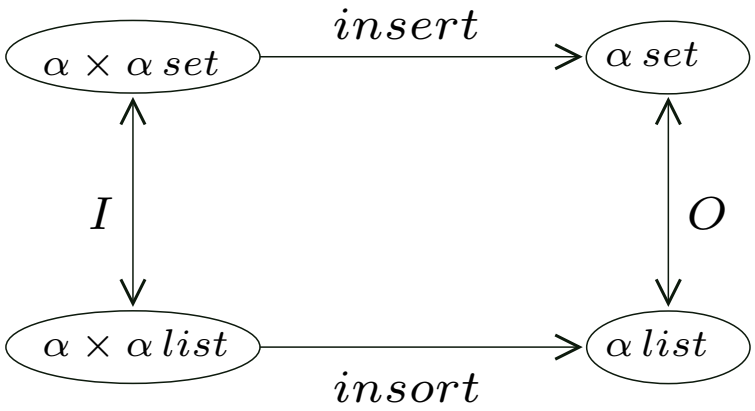
Representing Sets by Lists



Data Refinement for a Function

A simple example for refining a function:

Representing Sets by Lists



Can this be generalized to operations (i.e. "procedures" with input, output, and an implicit state transition) ?

Data Refinement

Principles of Data-Refinement

- Forward Simulation

Data Refinement

Principles of Data-Refinement

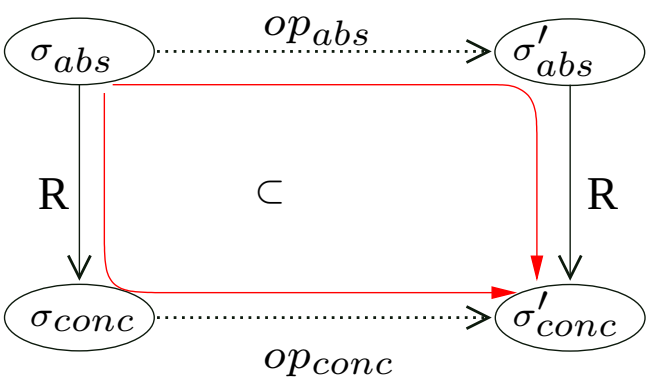
- Forward Simulation
- Backward Simulation

Data Refinement

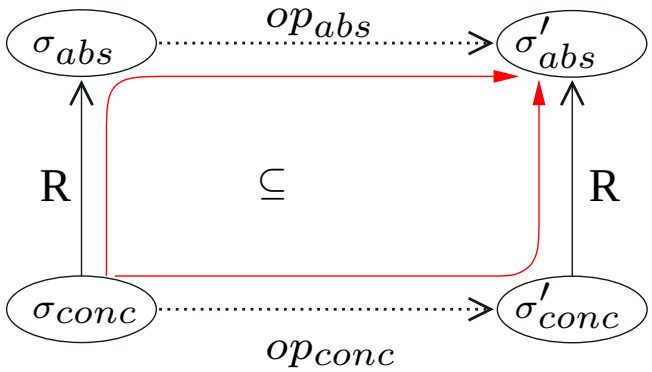
Principles of Data-Refinement

- Forward Simulation
- Backward Simulation

Forward Simulation



Backward Simulation

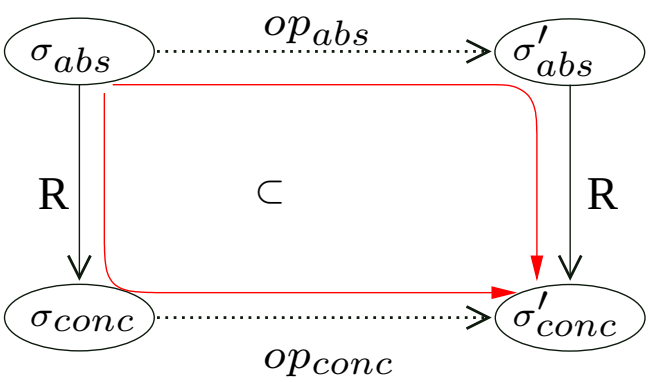


Data Refinement

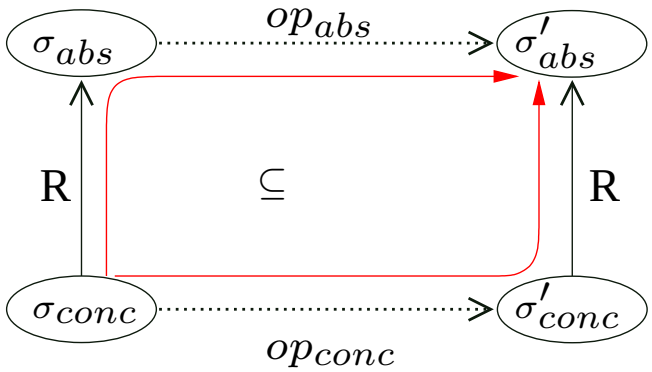
Principles of Data-Refinement

- Forward Simulation
- Backward Simulation

Forward Simulation



Backward Simulation



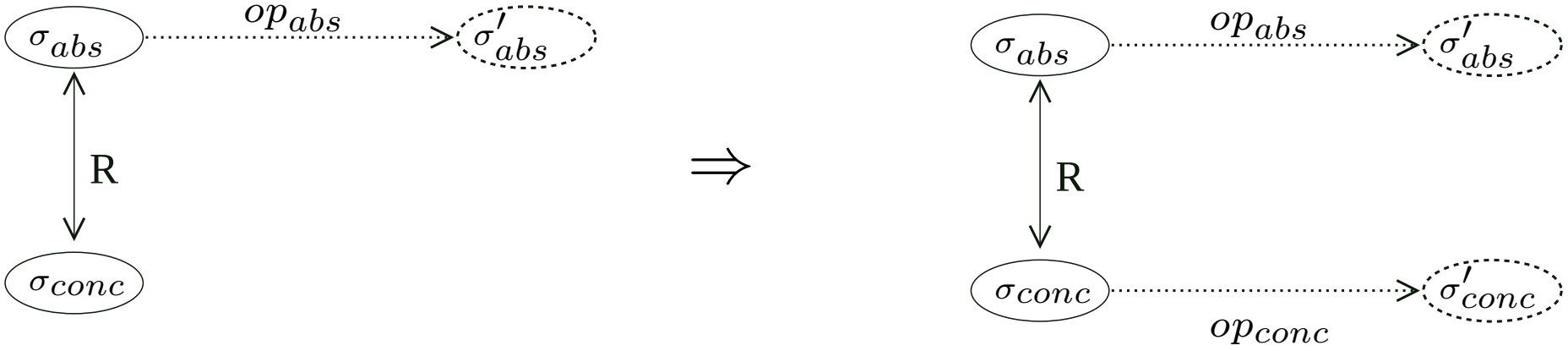
See also [Spi92] and [WD96]!

Data Refinement

Forward Simulation

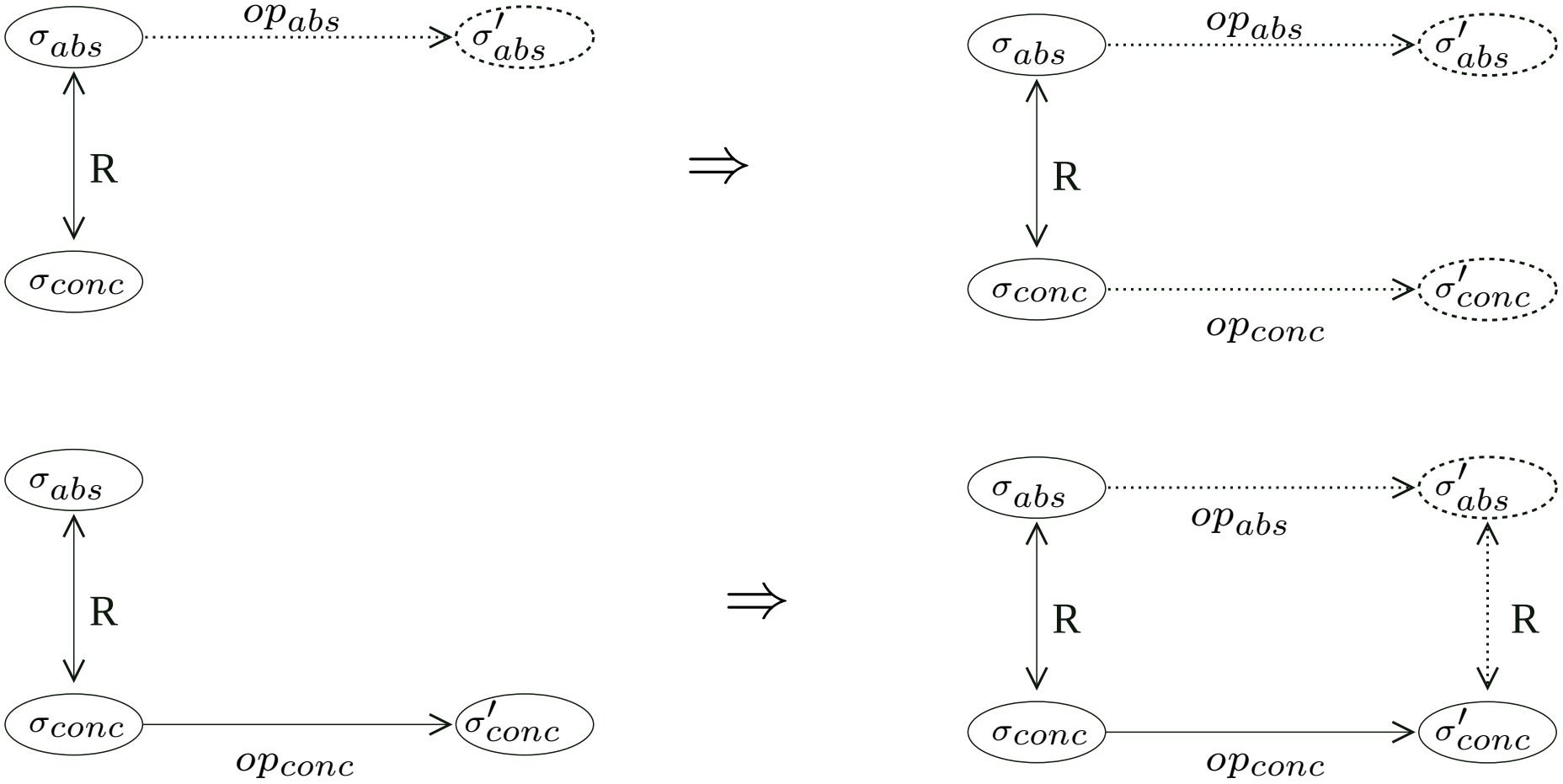
Data Refinement

Forward Simulation



Data Refinement

Forward Simulation



Data Refinement

Can we

Data Refinement

Can we

- represent refinement in Isabelle ?

Data Refinement

Can we

- represent refinement in Isabelle ?
- verify and compare refinement notions ?

Data Refinement

Can we

- represent refinement in Isabelle ?
- verify and compare refinement notions ?
- integrate refinement for functions and operations?

Data Refinement

Can we

- represent refinement in Isabelle ?
- verify and compare refinement notions ?
- integrate refinement for functions and operations?

YES! In the following, we present a theory of Abstract IOS Specifications and a forward simulation refinement on it. (backward refinement is analogously)

IOS-Forward Simulation

An abstract system IOS-step has the type:

IOS-Forward Simulation

An abstract system IOS-step has the type:

```
types ('i, 'o, 's) ios_rel = "(('i × 's) × ('o × 's)) set"
```

An Abstract IOS Specification is:
(closely related to a Z operation schema):

```
record ('i,'o,'s) spec =
  init :: "'s set"
  inv  :: "'s set"
  opn  :: "('i, 'o, 's) ios_rel"
```

IOS-Forward Simulation

The generalized abstraction relation on abstract IOS specifications looks as follows:

```
record ('i,'i',' o,'o',' s,'s') abs_rel =
  i   :: "('i × 'i') set"
  o   :: "('o × 'o') set"
  abs :: "('s × 's') set"
```

The relation is just a triple of relations on input data, output data and states.

IOS-Forward Simulation

We define a FS-refinement on IOS specifications by its three "proof obligations":

IOS-Forward Simulation

We define a FS-refinement on IOS specifications by its three "proof obligations":

constdefs

```
FS_refine :: " [('i,'o,'s) spec,
                ('i,'i','o,'o','s,'s') abs_rel ,
                ('i','o','s') spec] ⇒ bool"
A \<sqsubseteq>R C ≡
FS_init A R C ∧FS_corr1 A R C ∧FS_corr2 A R C
```

IOS-Forward Simulation

We define a FS-refinement on IOS specifications by its three "proof obligations":

constdefs

```
FS_refine :: " [('i,'o,'s) spec,
                ('i,'i','o,'o','s,'s') abs_rel ,
                ('i','o','s') spec] ⇒ bool"
A \<sqsubseteq>R C ≡
FS_init A R C ∧FS_corr1 A R C ∧FS_corr2 A R C
```

In conceptual notation, we will also write :$A \sqsubseteq_R^{fs} B$ for forward simulation (resp. $A \sqsubseteq_R^{bs} B$ for backward simulation).

IOS-Forward Simulation

The three conditions are:

IOS-Forward Simulation

The three conditions are:

- FS_init : The set of initial states must be compatible,
- FS_corr2: When an abstract state transition is possible, then a corresponding concrete state transition must be possible,
- FS_corr1: When a concrete operation reaches a target state, then the corresponding abstract must exist.

IOS-Forward Simulation

The three conditions are:

- FS_init : The set of initial states must be compatible,
- FS_corr2: When an abstract state transition is possible, then a corresponding concrete state transition must be possible,
- FS_corr1: When a concrete operation reaches a target state, then the corresponding abstract must exist.

(Terminology follows [WD96]).

IOS-Forward Simulation

The proof-obligation FS_init

IOS-Forward Simulation

The proof-obligation FS_init

$$
\begin{aligned}
&\mathsf{FS_init}\ \ \mathsf{A}\ \mathsf{R}\ \mathsf{C}\ \equiv \\
&\quad \forall\ \mathsf{cs} \in (\mathsf{inv}\ \ \mathsf{C}).\ \ \mathsf{cs} \in (\mathsf{init}\ \ \mathsf{C}) \longrightarrow \\
&\qquad\qquad\qquad \exists\ \mathsf{as} \in (\mathsf{inv}\ \ \mathsf{A}).\ \ \mathsf{as} \in (\mathsf{init}\ \ \mathsf{A}) \wedge (\mathsf{as}, \mathsf{cs}) \in \mathsf{abs}\ \mathsf{R}
\end{aligned}
$$

IOS-Forward Simulation

The proof-obligation FS_init

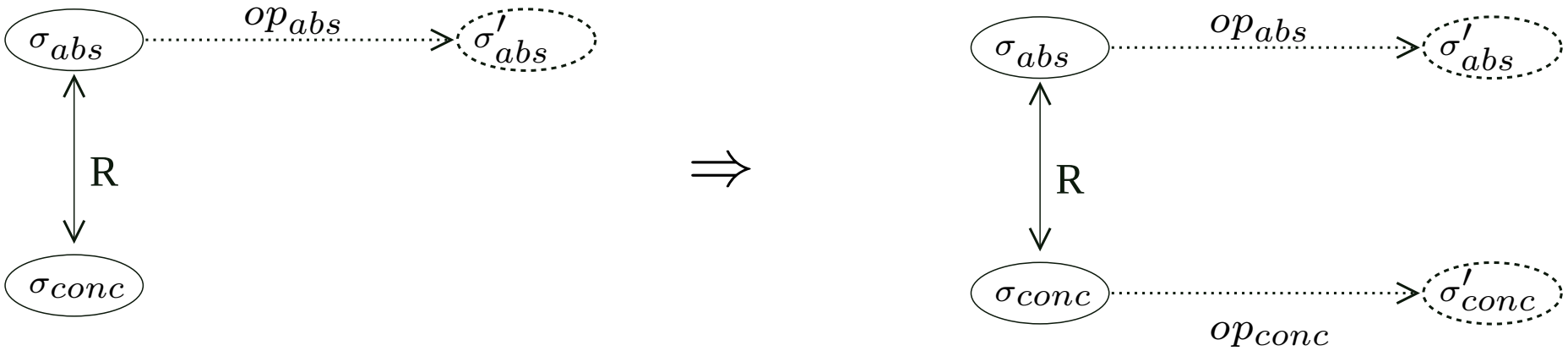
$$
\begin{aligned}
&\mathsf{FS_init}\ \ \mathsf{A}\ \mathsf{R}\ \mathsf{C}\ \equiv \\
&\quad \forall\ \mathsf{cs} \in (\mathsf{inv}\ \ \mathsf{C}).\ \ \mathsf{cs} \in (\mathsf{init}\ \ \mathsf{C}) \longrightarrow \\
&\qquad\qquad\qquad \exists\ \mathsf{as} \in (\mathsf{inv}\ \ \mathsf{A}).\ \ \mathsf{as} \in (\mathsf{init}\ \ \mathsf{A}) \wedge (\mathsf{as}, \mathsf{cs}) \in \mathsf{abs}\ \mathsf{R}
\end{aligned}
$$

IOS-Forward Simulation

Recall the diagrams for FS_corr2

IOS-Forward Simulation

Recall the diagrams for FS_corr2



IOS-Forward Simulation

The formalization for FS_corr2

IOS-Forward Simulation

The formalization for FS_corr2

$$
\begin{array}{l}
\text{FS_corr2 A R C} \equiv \\
\quad \forall \text{as} \in (\text{inv A}).\ \forall \text{cs} \in (\text{inv C}). \\
\quad \forall \text{inp} \in (\text{Domain(i R)}).\ \forall \text{inp'} \in (\text{Range(i R)}). \\
\qquad\qquad ((\text{inp}, \text{as}) \in \text{Domain (opn A)} \wedge \\
\qquad\qquad\ \ (\text{as}, \text{cs}) \in \text{abs R} \wedge (\text{inp}, \text{inp'}) \in \text{i R}) \\
\qquad\qquad \longrightarrow \\
\qquad\qquad\ \ (\text{inp'}, \text{cs}) \in \text{Domain(opn C)}
\end{array}
$$

IOS-Forward Simulation

The formalization for FS_corr2

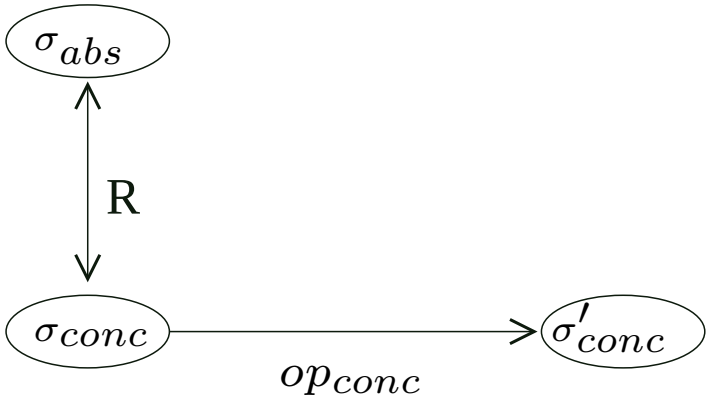
$$
\begin{array}{l}
\text{FS_corr2 A R C} \equiv \\
\quad \forall \text{as} \in (\text{inv A}).\ \forall \text{cs} \in (\text{inv C}). \\
\quad \forall \text{inp} \in (\text{Domain(i R)}).\ \forall \text{inp'} \in (\text{Range(i R)}). \\
\qquad\qquad ((\text{inp}, \text{as}) \in \text{Domain (opn A)} \wedge \\
\qquad\qquad\ \ (\text{as}, \text{cs}) \in \text{abs R} \wedge (\text{inp}, \text{inp'}) \in \text{i R}) \\
\qquad\qquad \longrightarrow \\
\qquad\qquad\ \ (\text{inp'}, \text{cs}) \in \text{Domain(opn C)}
\end{array}
$$

IOS-Forward Simulation

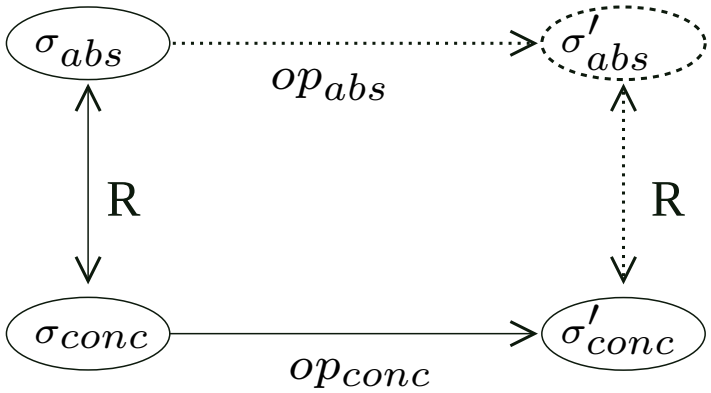
Recall the diagrams for FS_corr1

IOS-Forward Simulation

Recall the diagrams for FS_corr1



$\Rightarrow$



IOS-Forward Simulation

Recall the diagrams for FS_corr1

IOS-Forward Simulation

Recall the diagrams for FS_corr1

$$
\begin{array}{l}
\text{FS_corr1 A R C} \equiv \\
\quad \forall \text{as} \in (\text{inv A}).\ \forall \text{cs} \in (\text{inv C}).\ \forall \text{cs'} \in (\text{inv C}). \\
\quad \forall \text{inp} \in (\text{Domain(i R)}).\ \forall \text{inp'} \in (\text{Range(i R)}).\ \forall \text{out'} \in (\text{Range(o R)}). \\
\qquad\qquad ((\text{inp}, \text{as}) \in \text{Domain(opn A)} \wedge \\
\qquad\qquad\ (\text{as}, \text{cs}) \in \text{abs R} \wedge (\text{inp}, \text{inp'}) \in \text{i R} \wedge \\
\qquad\qquad\ ((\text{inp'}, \text{cs}), (\text{out'}, \text{cs'})) \in \text{opn C}) \\
\qquad\qquad \longrightarrow \\
\qquad\qquad (\exists\ \text{as'} \in (\text{inv A}).\ \exists\ \text{out} \in (\text{Domain(o R)}). \\
\qquad\qquad\qquad (\text{as'}, \text{cs'}) \in \text{abs R} \wedge (\text{out}, \text{out'}) \in \text{o R} \wedge \\
\qquad\qquad\qquad ((\text{inp}, \text{as}), (\text{out}, \text{as'})) \in \text{opn A})
\end{array}
$$

Tayloring IOS-Forward Simulation (1)

Tayloring forward simulations for functions :

Tayloring IOS-Forward Simulation (1)

Tayloring forward simulations for functions : Prerequisite:

We embed functions as abstract specifications:

Tayloring IOS-Forward Simulation (1)

Tayloring forward simulations for functions : Prerequisite:
We embed functions as abstract specifications:

constdefs

```
fun2op :: "['i set, 'i ⇒ 'o] ⇒ ('i,'o,unit) spec"
"fun2op precond F ≡(| init = {()}, inv = {()},
                      opn = {(a,b). ∃ x∈precond. a=(x,()) ∧
                                                b=(F x,())}|)"
```

Tayloring IOS-Forward Simulation (1)

Tayloring forward simulations for functions : Prerequisite:
We embed functions as abstract specifications:

constdefs

```
fun2op :: "['i set, 'i ⇒ 'o] ⇒ ('i,'o,unit) spec"
"fun2op precond F ≡(| init = {()}, inv = {()},
                      opn = {(a,b). ∃ x∈precond. a=(x,()) ∧
                                                b=(F x,())}|)"
```

procond serves as an additional means to formalize preconditions, under which the refinement is supposed to hold.

Tayloring IOS-Forward Simulation (1)

. . . derive the specialized version FS_refine_fun :

Tayloring IOS-Forward Simulation (1)

. . . derive the specialized version FS_refine_fun :

$[\![\; R = (\!| i = RI, o = RO, abs = Id |\!);$

$\forall$ inp $\in$ pa. A inp $\in$ Domain RO;
$\forall$ inp $\in$ pa. $\forall$ inp'. (inp, inp') $\in$ RI $\longrightarrow$ inp' $\in$ pc;
$\forall$ inp $\in$ pa. $\forall$ inp' $\in$ pc. (A inp, C inp') $\in$ RO $]\!]$

$\Longrightarrow$ (fun2op pa A) \<sqsubseteq>R (fun2op pc C)"

Tayloring IOS-Forward Simulation (1)

. . . derive the specialized version FS_refine_fun :

$$
\begin{array}{l}
[\![\ R = (\!|i = RI,\ o = RO,\ abs = Id|\!);\\
\quad \forall\ inp \in pa.\ A\ inp \in Domain\ RO;\\
\quad \forall\ inp \in pa.\ \forall\ inp'.\ (inp, inp') \in RI \longrightarrow inp' \in pc;\\
\quad \forall\ inp \in pa.\ \forall\ inp' \in pc.\ (A\ inp,\ C\ inp') \in RO\ ]\!]\\
\qquad \Longrightarrow \quad (fun2op\ pa\ A)\ \backslash<sqsubseteq>R\ (fun2op\ pc\ C)''
\end{array}
$$

Note that the first assumption constrains the structure of the generalized abstraction to default values on dummy states . . .

Tayloring IOS-Forward Simulation (1)

A (standard) example. We assume the usual:

Tayloring IOS-Forward Simulation (1)

A (standard) example. We assume the usual:

consts

```
insort     :: " ['a :: order , 'a list ] ⇒ 'a list "
is_sorted  :: " ['a list ] ⇒ bool"
```

Tayloring IOS-Forward Simulation (1)

A (standard) example. We assume the usual:

consts

```
insort     :: "['a::order, 'a list] ⇒ 'a list"
is_sorted  :: "['a list] ⇒ bool"
```

. . . and set up the refinement relation as:

consts

```
data_R     :: "('a::order set × 'a list)set"
set_list_R :: "('a::order × 'a set, 'a × 'a list,
                'a set, 'a list,
                unit, unit) abs_rel"
```

defs

data_R_def: "data_R $\equiv$ {(x, y). x=set y $\wedge$ is_sorted y}"

set_list_R_def :

" set_list_R $\equiv$ (| i = {(x,y). fst x = fst y $\wedge$ (snd x, snd y) $\in$ data_R},
o = data_R, abs = Id |)"

defs

data_R_def: "data_R $\equiv$ {(x, y). x=set y $\wedge$ is_sorted y}"

set_list_R_def :

" set_list_R $\equiv$ (| i = {(x,y). fst x = fst y $\wedge$ (snd x, snd y) $\in$ data_R},
o = data_R, abs = Id |)"

Tayloring IOS-Forward Simulation (1)

A refinement proof is started:

Tayloring IOS-Forward Simulation (1)

A refinement proof is started:

lemma insert_insort_refine_FS :

```
"(fun2op {λ(x,S). finite S}     (λ(x,S). insert x S))
 \<sqsubseteq>set_list_R
 (fun2op {λ(x,S). is_sorted S} (λ(x,S). insort x S))"
```

Tayloring IOS-Forward Simulation (1)

A refinement proof is started:

lemma insert_insort_refine_FS :

"(fun2op {λ(x,S). finite S} (λ(x,S). *insert* x S))
\<sqsubseteq>set_list_R
(fun2op {λ(x,S). is_sorted S} (λ(x,S). insort x S))"

. . . and, after applying FS_refine_fun as introduction rule, we derive the proof obligations:

Tayloring IOS-Forward Simulation (1)

A refinement proof is started:

lemma insert_insort_refine_FS :

"(fun2op {λ(x,S). finite S} (λ(x,S). *insert* x S))
\<sqsubseteq>set_list_R
(fun2op {λ(x,S). is_sorted S} (λ(x,S). insort x S))"

. . . and, after applying FS_refine_fun as introduction rule, we derive the proof obligations:

1. ∀ a b. finite b ⟶ (∃ y. *insert* a b = set y ∧ is_sorted y)
2. ∀ a b. finite b ⟶
   (∀ aa ba.
   is_sorted ba ⟶ *insert* a b = set (insort aa ba) ∧
   is_sorted (insort aa ba))

Tayloring IOS-Forward Simulation (2)

. . . derive FS_refine_opn_Z for operations

Tayloring IOS-Forward Simulation (2)

. . . derive FS_refine_opn_Z for operations

$$
\begin{array}{l}
[\![\ R = (\!|i = Id,\ o = Id,\ abs = Abs|\!);\\
\quad \forall cs \in (inv\ C).\ cs \in (init\ C) \longrightarrow\\
\qquad\qquad \exists\ as \in (inv\ A).\ as \in (init\ A) \wedge (as, cs) \in Abs;\\
\quad \forall as \in (inv\ A).\ \forall cs \in (inv\ C).\ \forall inp \in (Domain(i\ R)).\\
\qquad\qquad (\ pre(opn\ A)(inp, as) \wedge (as, cs) \in (abs\ R)) \longrightarrow\\
\qquad\qquad pre(opn\ C)(inp, cs);\\
\quad \forall as \in (inv\ A).\ \forall cs \in (inv\ C).\ \forall cs' \in (inv\ C).\ \forall inp.\ \forall out.\\
\qquad\qquad (\ pre(opn\ A)(inp, as) \wedge\\
\qquad\qquad\quad (as, cs) \in Abs \wedge ((inp, cs), (out, cs')) \in opn\ C) \longrightarrow\\
\qquad\qquad \exists\ as' \in (inv\ A).\ (as', cs') \in Abs \wedge\\
\qquad\qquad\qquad\qquad ((inp, as), (out, as')) \in (opn\ A))]\!]\\
\quad \Longrightarrow\ A\ \backslash <sqsubseteq> R\ C
\end{array}
$$

Tayloring IOS-Forward Simulation (2)

Do you recognize the pattern? :

Tayloring IOS-Forward Simulation (2)

Do you recognize the pattern? :This represents forward simulation a la [Spi92] and [WD96]):

Tayloring IOS-Forward Simulation (2)

Do you recognize the pattern? :This represents forward simulation a la [Spi92] and [WD96]):

$$\forall Cstate \bullet Cinit \rightarrow (\exists Astate \bullet Abs \wedge Ainit)$$

$$\forall Astate\, Cstate\, Cstate'\, x?\, y! \bullet$$
$$pre\, Aop \wedge Abs \wedge Cop \rightarrow (\exists Astate' \bullet Abs' \wedge Aop)$$
$$\forall Astate\, Cstate\, x? \bullet pre\, Aop \wedge Abs \rightarrow pre\, Cop$$

Note that in this refinement notion, input $x?$ and output $y!$ are identical!

Example: BirthdayBook Refinement

A (standard) example: Spivey's Birthdaybook[Spi92]:

Example: BirthdayBook Refinement

A (standard) example: Spivey's Birthdaybook[Spi92]:The states of the two systems are:

Example: BirthdayBook Refinement

A (standard) example: Spivey's Birthdaybook[Spi92]:The states of the two systems are:

```
record BirthdayBook =
  birthday :: "Name ~=> Date"
  known    :: "Name set"

record BirthdayBook1 =
  dates    :: "(nat ~=> Date)"
  hwm      :: nat
  names    :: "nat ~=> Name"
```

(The invariant states that known is equal to the domain of birthday).

Example: BirthdayBook Refinement

The two operation schemas are immediately represented as abstract IOS specifications:

consts

AddBirthday :: "((Name ×Date), unit, BirthdayBook) spec"
AddBirthday1:: "((Name ×Date), unit, BirthdayBook1) spec"
. . .

Example: BirthdayBook Refinement

The abstraction relation between the underlying states is:

Example: BirthdayBook Refinement

The abstraction relation between the underlying states is:

constdefs

```
  Abs :: "(BirthdayBook ×BirthdayBook1) set"
"Abs ≡ {(x,y).((known x) = {n. ∃ i∈{1..(hwm y)}.
                                  n = the (names y i)}) ∧
                (∀ i∈{1..(hwm y)}. birthday x (the(names y i))
                                  = dates y (the(names y i)))}"
```

Example: BirthdayBook Refinement

The abstraction relation between the underlying states is:

constdefs

```
  Abs :: "(BirthdayBook ×BirthdayBook1) set"
"Abs ≡ {(x,y).((known x) = {n. ∃ i∈{1..(hwm y)}.
                                  n = the (names y i)}) ∧
                (∀ i∈{1..(hwm y)}. birthday x (the(names y i))
                                  = dates y (the(names y i)))}"
```

. . . which is generalized to:

Example: BirthdayBook Refinement

The abstraction relation between the underlying states is:

constdefs

```
  Abs :: "(BirthdayBook ×BirthdayBook1) set"
 "Abs ≡ {(x,y).((known x) = {n. ∃ i∈{1..(hwm y)}.
                                  n = the (names y i)}) ∧
                 (∀ i∈{1..(hwm y)}. birthday x (the(names y i))
                                  = dates y (the(names y i)))}"
```

. . . which is generalized to:

constdefs

```
  gen_Abs :: "('a,'a,'b,'b,BirthdayBook,BirthdayBook1) abs_rel"
 "gen_Abs ≡ (|i = Id, o = Id, abs = Abs|)"
```

Example: BirthdayBook Refinement

The question to be asked:

Example: BirthdayBook Refinement

The question to be asked:

lemma AddBrithday_FS_refine :

"AddBirthday \<sqsubseteq>gen_Abs AddBirthday1"

Example: BirthdayBook Refinement

Applying FS_refine_opn_Z yields:

Example: BirthdayBook Refinement

Applying FS_refine_opn_Z yields:

1. $\forall$ cs$\in$spec.inv AddBirthday1.
   cs $\in$ init AddBirthday1 $\longrightarrow$
   ($\exists$ as$\in$inv AddBirthday. as$\in$init AddBirthday $\wedge$(as, cs)$\in$Abs)
2. $\forall$ as$\in$inv AddBirthday. $\forall$ cs$\in$inv AddBirthday1. $\forall$ inp.
   pre(opn AddBirthday)(inp,as) $\wedge$ (as, cs)$\in$Abs $\longrightarrow$
   pre(opn AddBirthday1)(inp,cs)
3. $\forall$ as$\in$inv AddBirthday. $\forall$ cs$\in$inv AddBirthday1.
   $\forall$ cs'$\in$inv AddBirthday1. $\forall$ inp out.
   pre(opn AddBirthday)(inp,as) $\wedge$
   (as, cs)$\in$Abs $\wedge$((inp, cs), out, cs')$\in$opn AddBirthday1 $\longrightarrow$
   $\exists$ as'$\in$inv AddBirthday.
   (as', cs')$\in$Abs $\wedge$((inp, as), out, as')$\in$opn AddBirthday

(see [Spi92] and the HOL-Z-disribution [BRW03a]!)

Connection to Behavioral Refinement(1)

- How do abstract IOS specifications relate to behavioral models?

Connection to Behavioral Refinement(1)

- How do abstract IOS specifications relate to behavioral models?
- Can we extend reasoning over refinements of individual system steps to sequences of steps ?

Connection to Behavioral Refinement(1)

- How do abstract IOS specifications relate to behavioral models?
- Can we extend reasoning over refinements of individual system steps to sequences of steps ?
- How do established notions of behavioral specification relate to forward/backward simulation ?

Connection to Behavioral Refinement(1)

- How do abstract IOS specifications relate to behavioral models?
- Can we extend reasoning over refinements of individual system steps to sequences of steps ?
- How do established notions of behavioral specification relate to forward/backward simulation ?

Partial Answer: abstract IOS specifications generate behavioral notions like Kripke-Structures, (Event) Traces and (Event) Failures. The former talks about states, the latter two over "observable input/output"(=Events)

Connection to Behavioral Refinement(1)

State Projection into Kripke Structures :

Connection to Behavioral Refinement(1)

State Projection into Kripke Structures :

types

```
  's trace = "nat ⇒ 's"

record 's kripke =
  init    :: "'s set"
  step    :: "('s × 's) set"
```

constdefs

```
  state_projection :: "('i,'o,'s) spec ⇒ 's kripke"
 " state_projection A ≡
   (|kripke.init = spec.init A,
    kripke.step = {(s1,s2). ∃ i' o'.((i',s1),(o',s2))∈spec.opn A}|)"
```

Connection to Behavioral Refinement(1)

constdefs

```
  is_trace      :: "['s kripke, 's trace] => bool"
" is_trace K t ≡ t 0 ∈ kripke.init K ∧
                 (∀i. (t i, t (Suc i)) ∈ kripke.step K)"
  traces        :: "'s kripke => 's trace set"
" traces K      ≡ { t. is_trace K t }"
```

Connection to Behavioral Refinement(1)

constdefs

```
  is_trace        :: "['s kripke, 's trace] => bool"
" is_trace K t ≡ t 0 ∈ kripke.init K ∧
                 (∀i. (t i, t (Suc i)) ∈ kripke.step K)"
  traces          :: "'s kripke => 's trace set"
" traces K      ≡ { t. is_trace K t }"
```

And now, a standard temporal logics $K \models \phi$ can be defined on top of the Kripke structure K. Open problem: Under which conditions can a forward refinement allow for system abstractions?

```
⟦ A \<sqsubseteq>R C; kripke_projection A |= phi ⟧
⟹  kripke_projection C |= phi
```

Paves the way for temporal abstractions and model-checking.

Connection to Behavioral Refinement(2)

Event Projection of ('i,'o,'s)spec's to event traces:

Connection to Behavioral Refinement(2)

Event Projection of ('i,'o,'s)spec's to event traces:

constdefs

```
  is_gen_trace     ::[('i,'o,'s)spec,(('i×'s)×('o×'s))trace] ⇒ bool
" is_gen_trace  A t ≡(snd(fst(t 0)) ∈ spec. init A ∧
                        (∀i. t i ∈ spec.opn A) ∧
                        (∀i. snd(snd(t i))= snd(fst(t (Suc i)))))"
  gen_traces     :: ('i,'o,'s) spec ⇒ (('i×'s)×('o×'s))trace set
" gen_traces A ≡ { t.   is_gen_trace A t }"

  event_traces_projection   :: ('i,'o,'s) spec ⇒ ('i×'o)trace set
"  event_traces_projection  A  ≡(λf n.(λ(x,y). (fst x, fst y))(f n))
                                                ` (gen_traces A)"
```

In order to accomodate ('i,'o,'s)spec for CSP-processes, for example, states 's must be instantiated with process terms, and opn by transition of operational CSP semantics [A.W97] ...

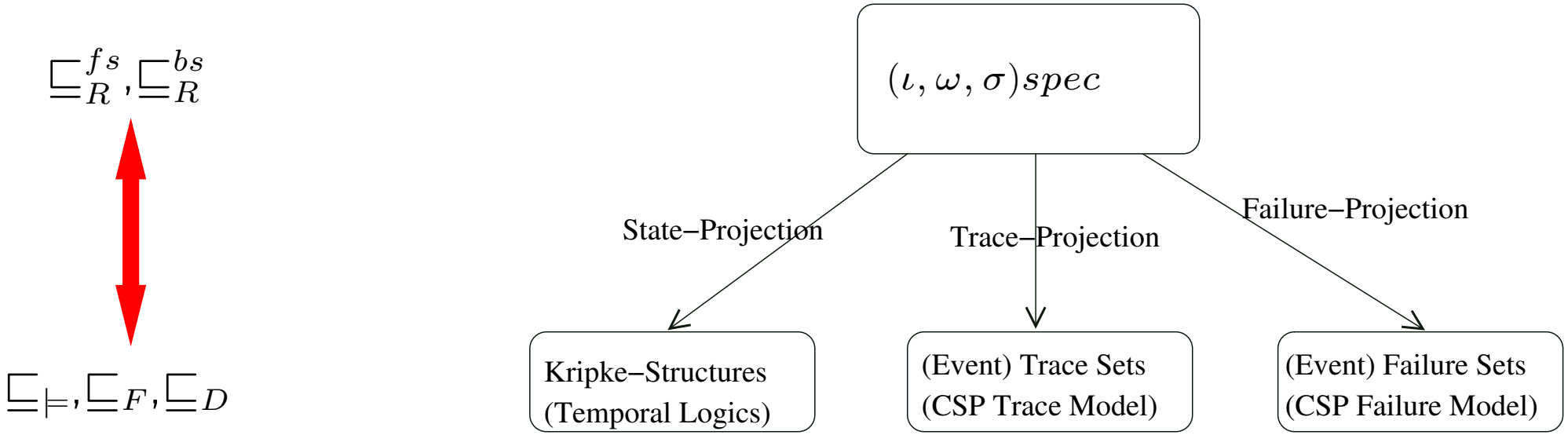
In order to accomodate ('i,'o,'s)spec for CSP-processes, for example, states 's must be instantiated with process terms, and opn by transition of operational CSP semantics [A.W97] ...

Open problem: Under which conditions allows fs-refinement simplifying process-refinement, i.e. to inclusion of trace or failure sets?

Connection to Behavioral Refinement(3)

Overview:

Abstract Specifications and their Behavioral Models



Summary

- Refinement can be represented in a generalized framework (such as IOS-specifications)
- Approach can be used for data refinement and behavioral refinement as well
- Approach can be used for proving meta-theoretic properties of refinements (reflexive?, transitive?, composable?) too
- Approach can be used for automated proof support.

Higher-Order Logic Applications: HOL-OCL

Achim D. Brucker

Overview

- Motivation
- An Introduction to UML/OCL
- Formalizing Class Diagrams
- Excursus: Defining Semantics
- Embedding OCL into Isabelle/HOL
- Conclusion

Motivation

The Situation Today: A Software Engineering Problem

- Software systems are
  - getting more and more complex.
  - used in safety and security critical applications.
- We think:
  - Complex software systems require a precise specification of its architecture and components.
  - Semi-formal methods (like UML diagrams) are not strong enough.

Specification should be useful, i.e. not only documentation!

Why use Formal Methods in Software Development

There are many reasons for using formal methods:

- safety critical applications, e.g. flight or railway control.
- security critical applications, e.g. access control.
- financial reasons (e.g. warranty), e.g. embedded devices.
- legal reasons, e.g. certifications.

Many successful applications of formal methods proof their success!

Why Formal Methods are not widely accepted in software industry?

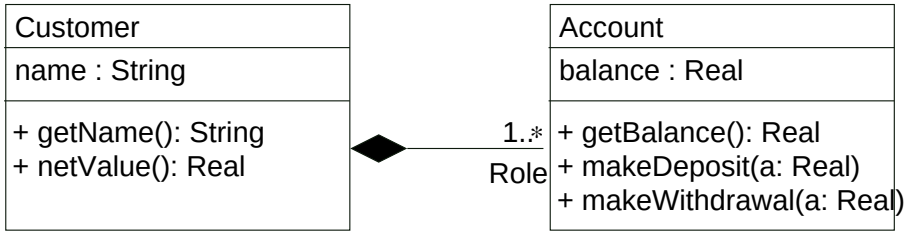
- Only a few formal methods address industrial needs:
  - support for object-oriented modeling and programming.
  - mainly automatic (?).
  - integration in standard CASE tools and processes.
- Formal methods people and industrial software developer are often speaking different languages.

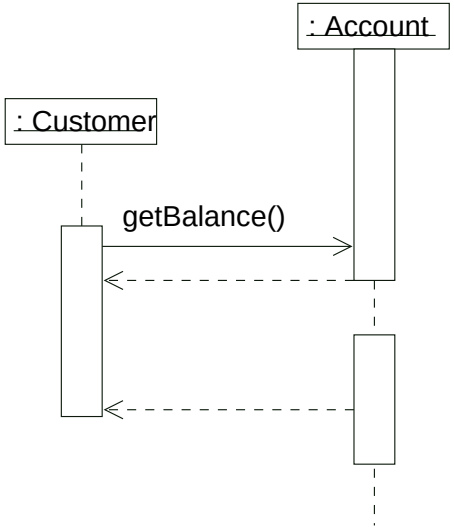
To tackle these challenges we provide a a formal foundation for (supporting object-orientation) for a industrial accepted specification languages (UML/OCL) [omg01, omg03].

An Introduction to UML/OCL

The Unified Modeling Language (UML)

- visual modeling language
- many diagram types, e.g.
  - class diagrams (static)
  - state charts (dynamic)
  - use cases
- object-oriented development
- industrial tool support
- OMG standard with semi-formal semantics





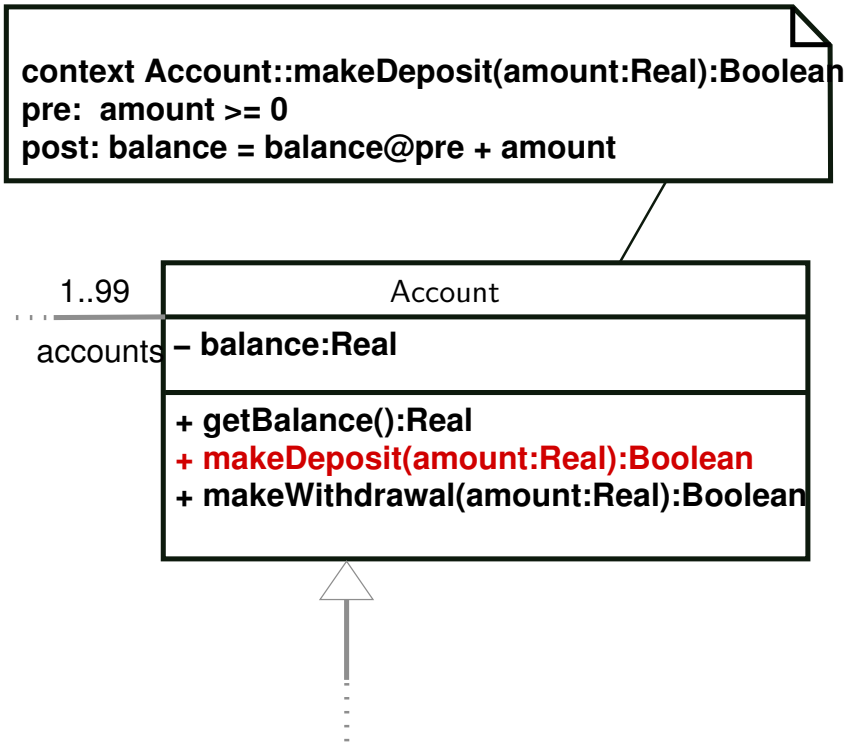
Are UML diagrams enough to specify OO systems formally?

- The short answer:
  - UML diagrams are not powerful enough for supporting formal reasoning over specifications.
- The long answer:

  We want to be able to
  - verify (proof) properties
  - refine specifications
- Thus we need:
  - a formal extension of UML.

The Object Constraint Language (OCL)

- based on first-order logic with equality and typed set theory
- designed for annotating UML diagrams
- in the context of class–diagrams:
  - preconditions
  - postconditions
  - invariants
- can be used for other diagrams too (not discussed here)



OCL — A Simple Examples

- "Uniqueness" constraint for the class Account:

```
context Account inv:
    Account.allInstances->forAll(a1,a2 | a1.id = a2.id
                                        implies a1 = a2)
```

- Properties of the class diagram can be described, e.g. multiplicities:

```
context Account inv: Account.owner->size = 1
```

- Meaning of the method makeDeposit():

```
context Account::makeDeposit(amount:Real):Boolean
pre: amount >= 0
post: balance = balance@pre + amount
```

OCL keywords Path expressions from UML model

Challenges of Formalizing UML/OCL

Only few formal methods are specialized for analyzing object oriented specifications.

- Problems and open questions:
  - object equality and aliasing
  - embedding of object structures into logics
  - referencing and dereferencing, including "null" references
  - dynamic binding
  - polymorphism
  - representing object-oriented concepts inside λ-calculi
  - providing a (suitable, shallow) representation in theorem provers
  - . . .

How to proceed

For Turning UML/OCL into a formal method we need

1. a formal foundation of UML class diagrams.
   - typed path expressions
   - inheritance
   - . . .
2. a formal semantics of OCL and proof support for OCL.
   - reasoning over UML path expressions
   - large libraries
   - . . .

Formalizing Class Diagrams

The Challenges of UML

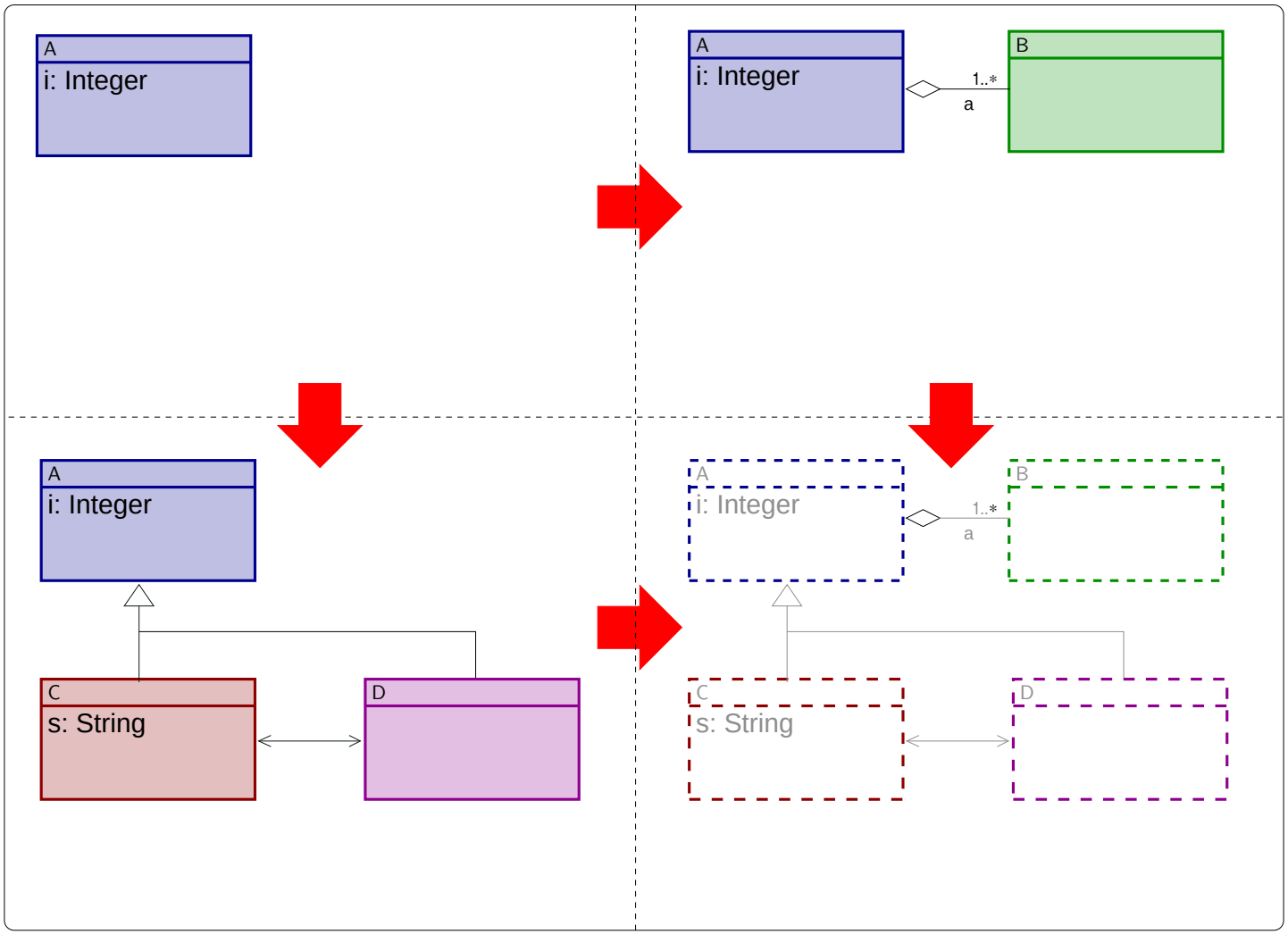
1. typed path expressions
2. typed object store
3. inheritance, thus extendibility in
   - data (by subtyping)
   - methods (late binding)
4. dynamic vs. static types
5. casting types

Typed Objects

Problem: • How to efficiently represent the types of objects?

What is the meaning of path expressions?

• How to represent the underlying state ?


A
i: Integer
A
i: Integer
1..*
a
B
A
i: Integer
C
s: String
D
A
i: Integer
1..*
a
B
C
s: String
D

Typed Objects

Problem: how to efficiently represent the types of objects

Typed Objects

Problem: how to efficiently represent the types of objects

Answer: Representing Class-Extensions by

1. a unique type key

Typed Objects

Problem: how to efficiently represent the types of objects

Answer: Representing Class-Extensions by

1. a unique type key
2. a type extension
   (just the product of the type key and the attributes of this extension)

Typed Objects

Problem: how to efficiently represent the types of objects

Answer: Representing Class-Extensions by

1. a unique type key
2. a type extension
   (just the product of the type key and the attributes of this extension)
3. its static type.
4. tests over the content of the extension field
   (constituting the dynamic type)

Typed Objects

Problem: how to efficiently represent the types of objects

Answer: Representing Class-Extensions by

1. a unique type key
2. a type extension
   (just the product of the type key and the attributes of this extension)
3. its static type.
4. tests over the content of the extension field
   (constituting the dynamic type)

5. projections and injections

5. projections and injections
6. develop mechanical support for injections, projections, and test theorems

5. projections and injections
6. develop mechanical support for injections, projections, and test theorems
7. generate UML path-syntax.

Typed Objects: Static Types

The static type of an object of a class is a product:

Typed Objects: Static Types

The static type of an object of a class is a product:

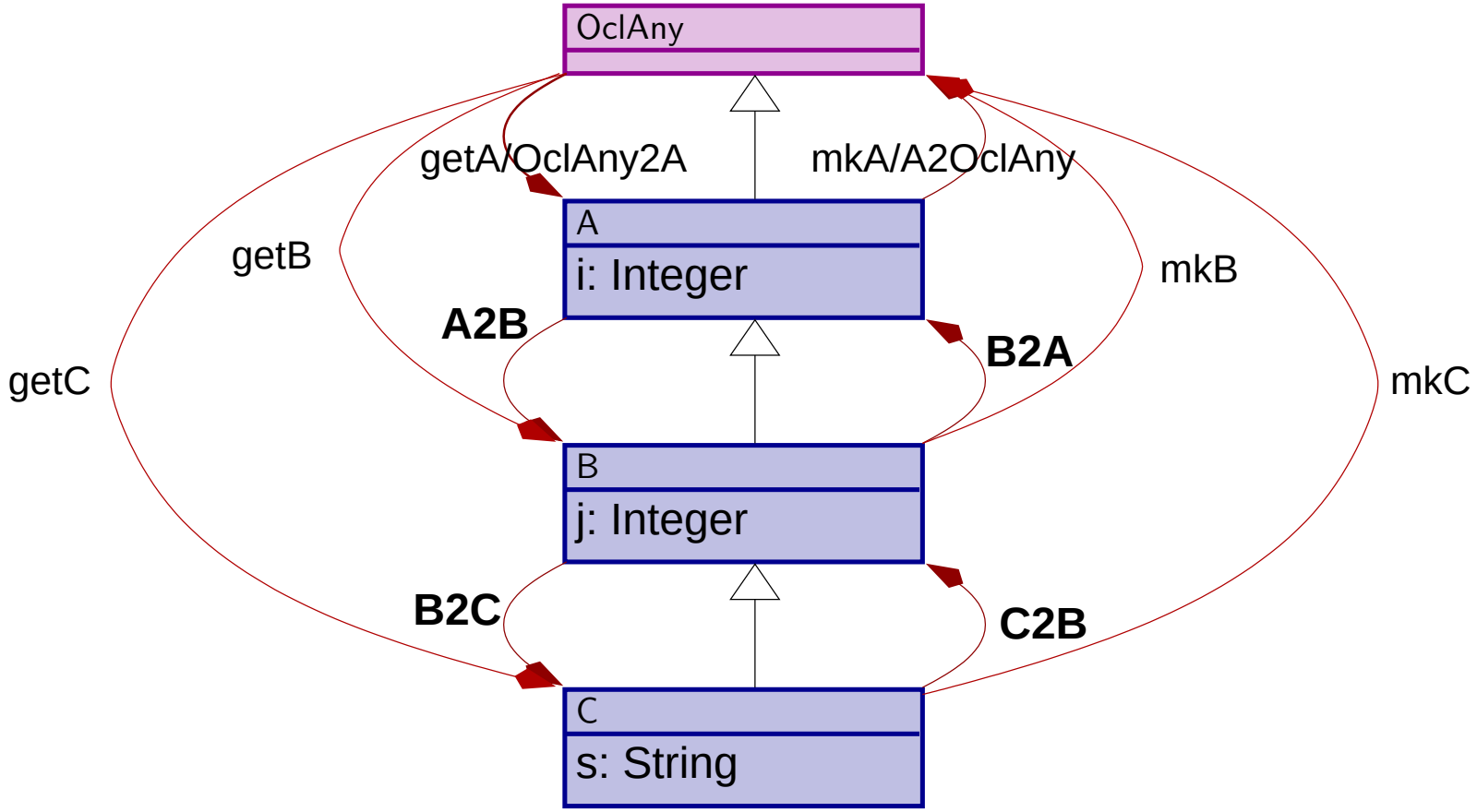
1. over all father class extensions (simplified)
   (Example A: $OclAny_key \times (A_key \times \alpha))$

Typed Objects: Static Types

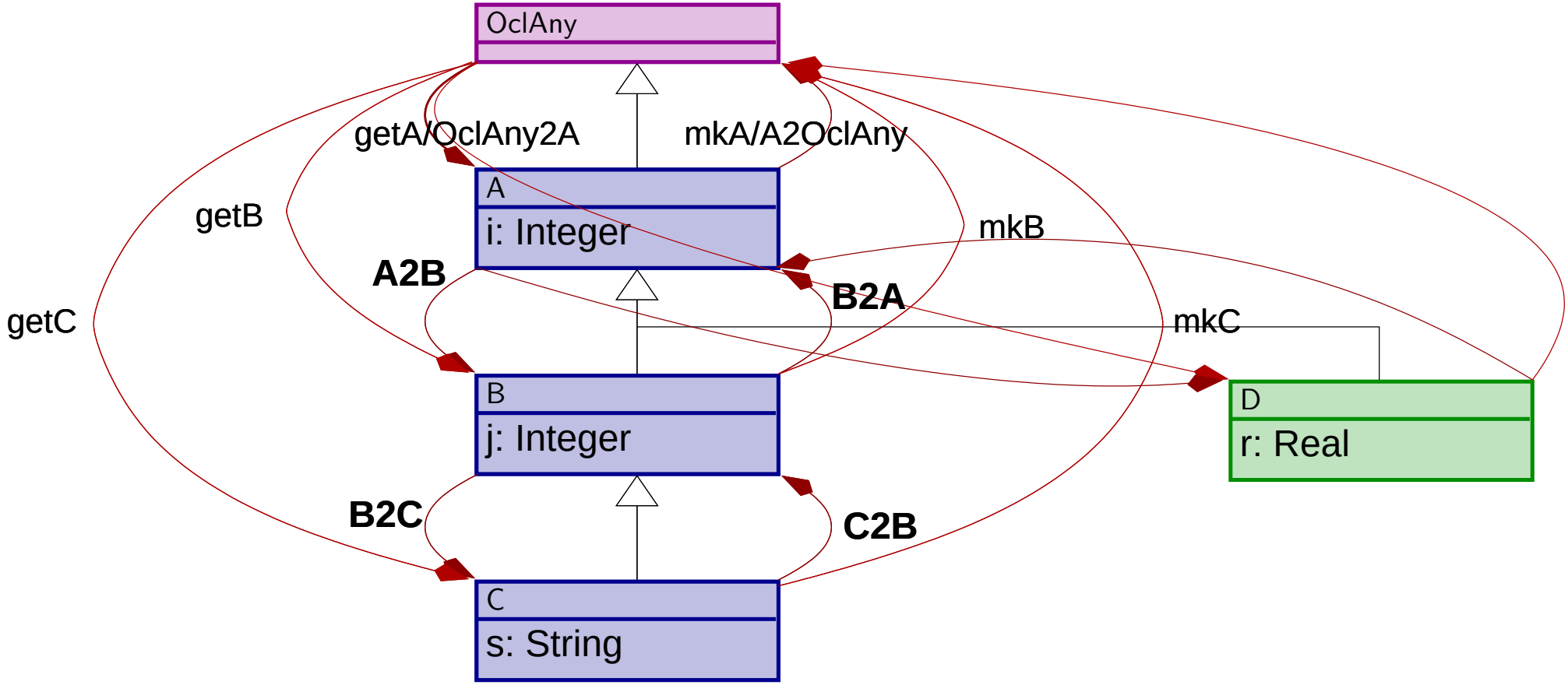
The static type of an object of a class is a product:

1. over all father class extensions (simplified)
   (Example A: $OclAny_key \times (A_key \times \alpha))$
2. and an extension field α:
   (ranging over the sum of future extensions:
   $B_1 + \cdots + B_n + \beta$)

Typed Objects: Dynamic Types



Typed Objects: Dynamic Types



Automatically Derived Properties on Objects

1. dynamic tests, injections and projections
   (is_A(X), "castings" for $\circ$ clAsType)

Automatically Derived Properties on Objects

1. dynamic tests, injections and projections
   (is_A(X), "castings" for $\circ$ clAsType)
2. distinctness theorems
   (dynamic B of static A is different from dynamic D of static A)

Automatically Derived Properties on Objects

1. dynamic tests, injections and projections
   (is_A(X), "castings" for $\circ$ clAsType)
2. distinctness theorems
   (dynamic B of static A is different from dynamic D of static A)
3. inclusion theorems
   (all dynamic B of static A are dynamic A of static A . . .)

Automatically Derived Properties on Objects

1. dynamic tests, injections and projections
   (is_A(X), "castings" for $\circ$ clAsType)
2. distinctness theorems
   (dynamic B of static A is different from dynamic D of static A)
3. inclusion theorems
   (all dynamic B of static A are dynamic A of static A . . .)
4. exhaustion theorems
   (A objects are void or have B extension or have D extension or other extension)

Automatically Derived Properties on Objects

1. dynamic tests, injections and projections
   (is_A(X), "castings" for $\circ$ clAsType)
2. distinctness theorems
   (dynamic B of static A is different from dynamic D of static A)
3. inclusion theorems
   (all dynamic B of static A are dynamic A of static A . . .)
4. exhaustion theorems
   (A objects are void or have B extension or have D extension or other extension)

5. invariance theorems

$$(\ \mathrm{DEF}(A::A) \Longrightarrow \mathrm{INV_A})$$

Open question: induction theorems.

Summary

In this representation,

1. for a static type, any dynamic extension will be typesafe accepted
   (by Isabelle/HOL; even if we do not know how many we have: "open world")

Summary

In this representation,

1. for a static type, any dynamic extension will be typesafe accepted
   (by Isabelle/HOL; even if we do not know how many we have: "open world")
2. their is no need for "wellformedness"-predicates of objects

Summary

In this representation,

1. for a static type, any dynamic extension will be typesafe accepted
   (by Isabelle/HOL; even if we do not know how many we have: "open world")
2. their is no need for "wellformedness"-predicates of objects
3. conversions between static types must be explicit
4. there is a uniform sum-type that comprises them all class extensions

Summary

In this representation,

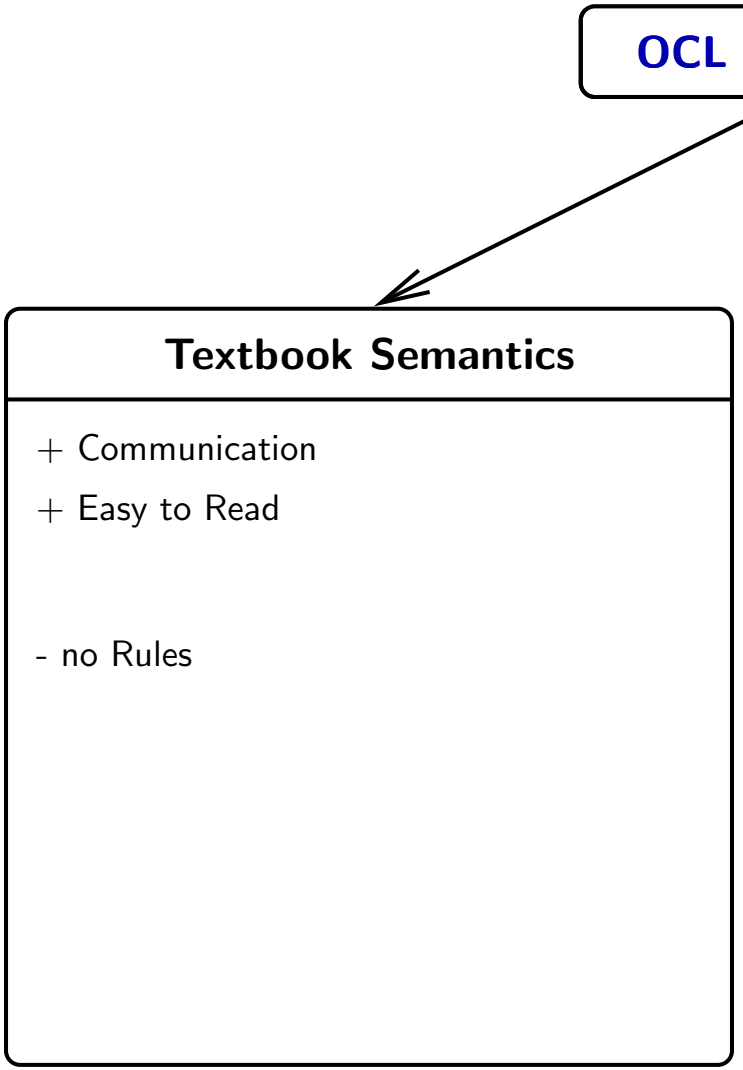
1. for a static type, any dynamic extension will be typesafe accepted
   (by Isabelle/HOL; even if we do not know how many we have: "open world")
2. their is no need for "wellformedness"-predicates of objects
3. conversions between static types must be explicit
4. there is a uniform sum-type that comprises them all class extensions
5. proofs remain valid under extension

Excursus: Defining Semantics

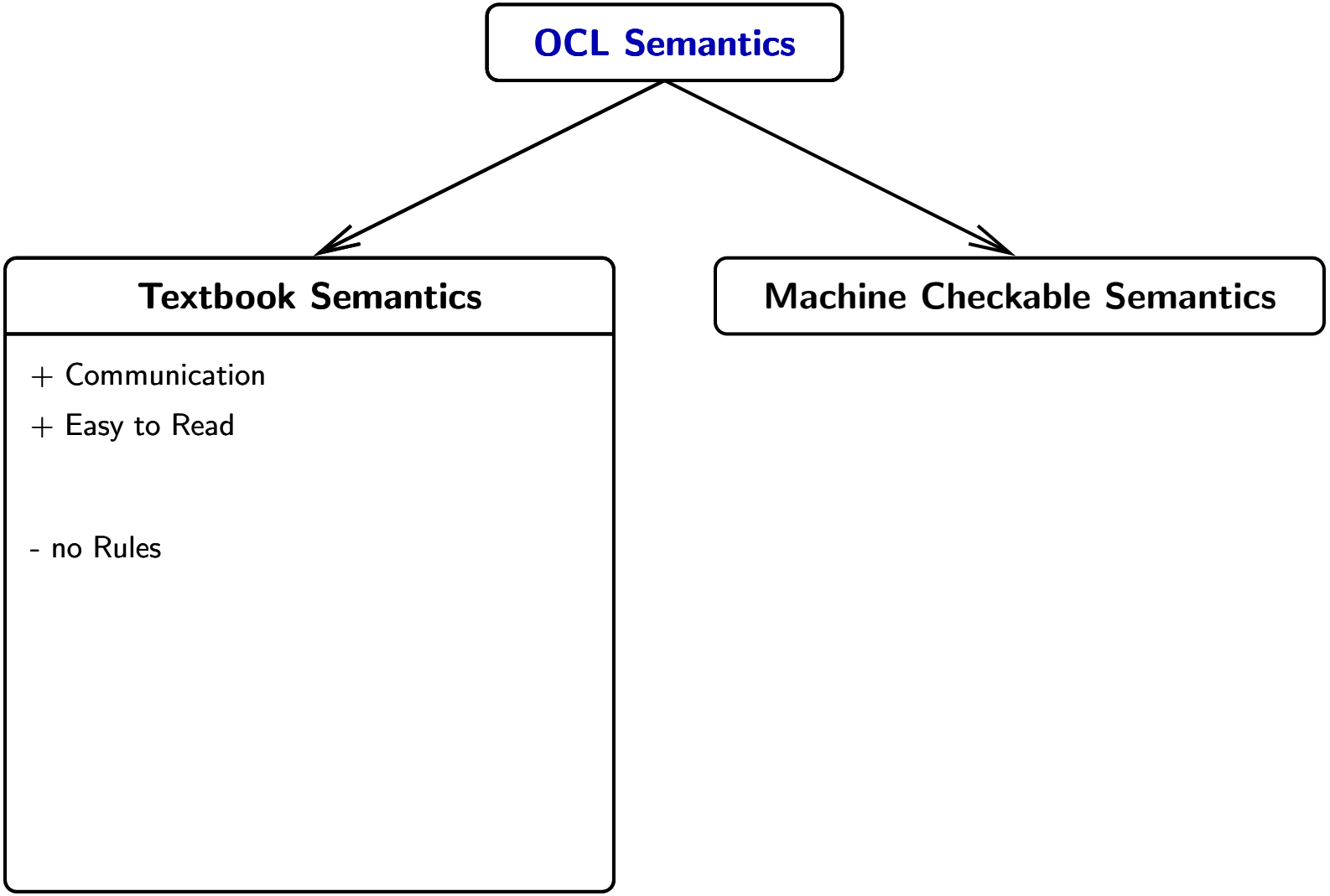
Semantics

OCL Semantics

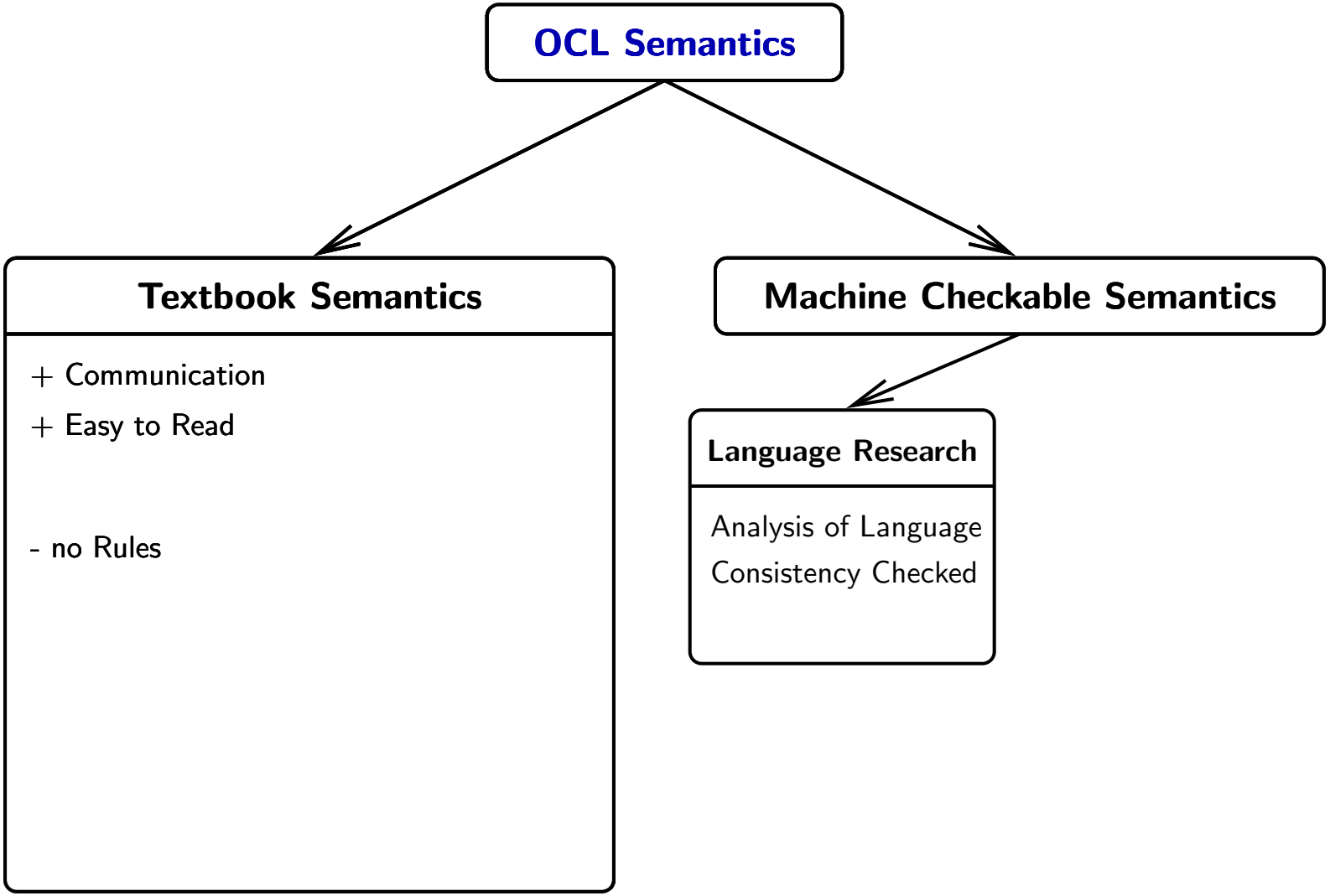
Semantics


OCL Semantics
Textbook Semantics
+ Communication
+ Easy to Read
- no Rules

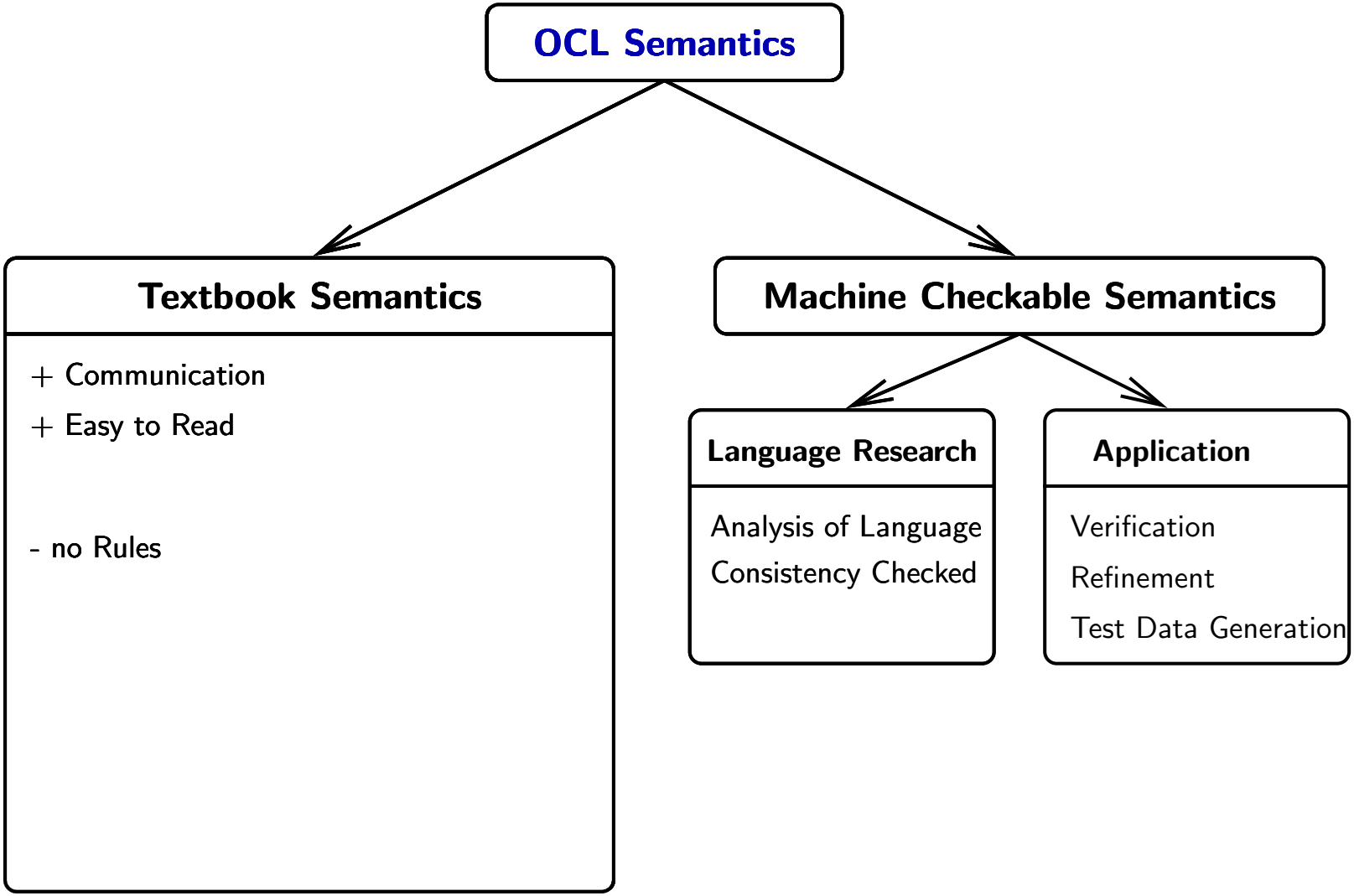
Semantics


OCL Semantics
Textbook Semantics
+ Communication
+ Easy to Read
- no Rules
Machine Checkable Semantics

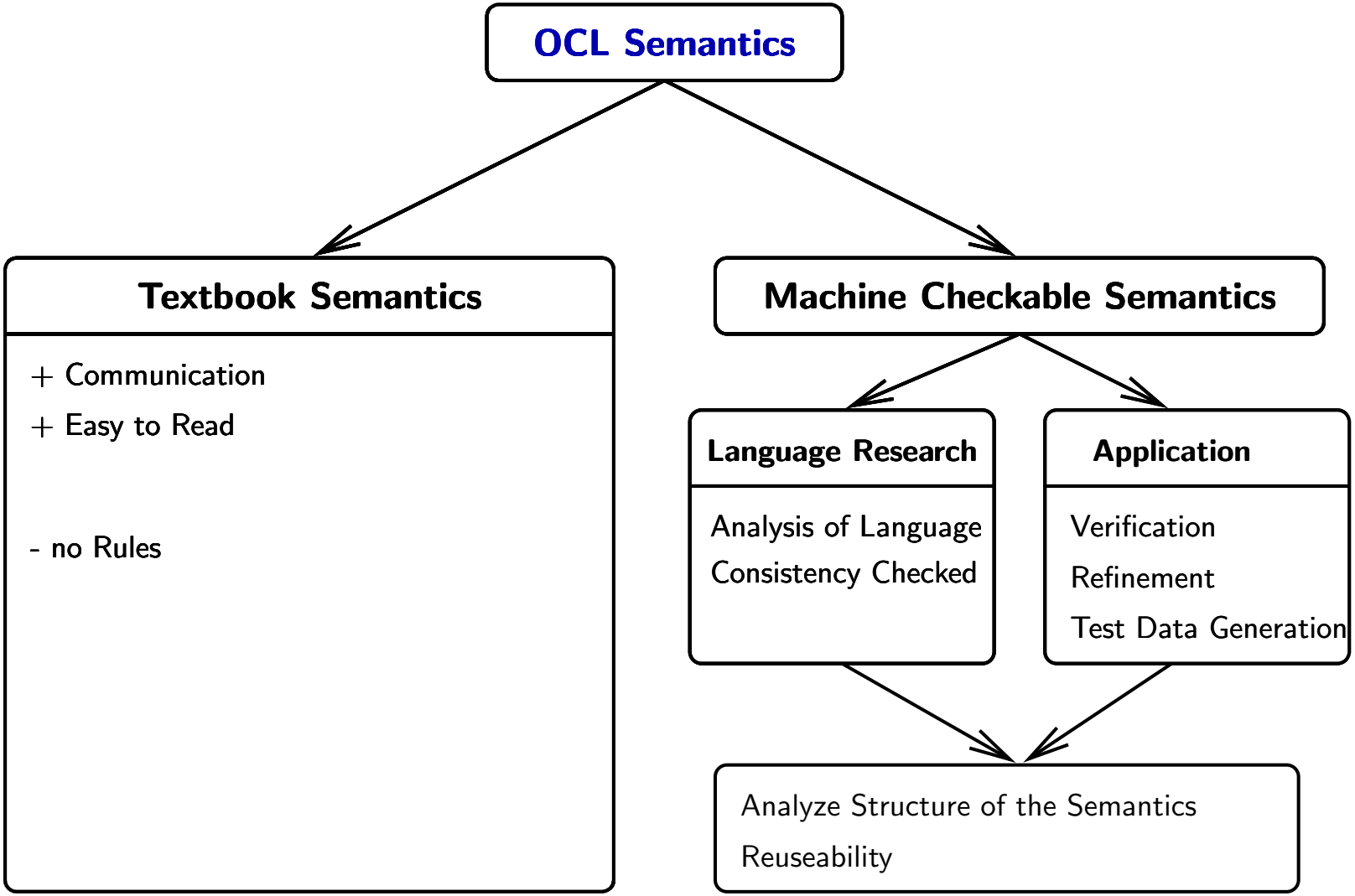
Semantics


OCL Semantics
Textbook Semantics
+ Communication
+ Easy to Read
- no Rules
Machine Checkable Semantics
Language Research
Analysis of Language
Consistency Checked

Semantics


OCL Semantics
Textbook Semantics
+ Communication
+ Easy to Read
- no Rules
Machine Checkable Semantics
Language Research
Analysis of Language
Consistency Checked
Application
Verification
Refinement
Test Data Generation

Semantics


OCL Semantics
Textbook Semantics
+ Communication
+ Easy to Read
- no Rules
Machine Checkable Semantics
Language Research
Analysis of Language
Consistency Checked
Application
Verification
Refinement
Test Data Generation
Analyze Structure of the Semantics
Reuseability

Textbook Semantics: An Example

- The interpretation of the logical and is given by a truth-table:

a	b	a and b
$false$	$false$	$false$
$false$	$true$	$false$
$false$	$\perp_{\mathscr{L}}$	$false$

a	b	a and b
$true$	$false$	$false$
$true$	$true$	$true$
$true$	$\perp_{\mathscr{L}}$	$\perp_{\mathscr{L}}$

a	b	a and b
$\perp_{\mathscr{L}}$	$false$	$false$
$\perp_{\mathscr{L}}$	$true$	$\perp_{\mathscr{L}}$
$\perp_{\mathscr{L}}$	$\perp_{\mathscr{L}}$	$\perp_{\mathscr{L}}$

- The Interpretation of "X->union(Y)" for sets ("$X \cup Y$"):

$$I(\cup)(X,Y) \equiv \begin{cases} X \cup Y & \text{if } X \neq \perp_{\mathscr{L}} \text{ and } Y \neq \perp_{\mathscr{L}} \\ \perp_{\mathscr{L}} & \text{otherwise} \end{cases}$$

This is a strict and lifted version of the union of "mathematical sets".

Textbook Semantics

- "Paper-and-Pencil" work in mathematical notation.

+ Useful to communicate semantics.

+ Easy to read.

- No rules, no laws.

- Informal or meta-logic definitions
  (*"The Set is the mathematical set."*).

- It is easy to write inconsistent semantic definitions.

Machine-Checkable Semantics

Motivation: Honor the semantical structure of the language.

- A machine-checked semantics
  - conservative embeddings guarantee consistency of the semantics.
  - builds the basis for analyzing language features.
  - allows incremental changes of semantics.
- As basis of further tool support for
  - reasoning over specifications.
  - refinement of specifications.
  - automatic test data generation.

Shallow vs. Deep Embeddings

Representing the logical operations or and and via a

- shallow embedding:
  Direct definition of the semantics, e.g. each construct is represented by some function on a semantic domain.

$$x\mathsf{and}y \equiv \lambda e.\ .x\ e \wedge y\ e \qquad xory \equiv \lambda e.\ .x\ e \vee y\ e$$

- deep embedding:
  The abstract syntax is presented as a datatype and a semantic function I from syntax to semantics.

$$expr = \text{var } var \mid expr \text{ and } expr \mid expr \text{ or } expr$$

and the explicit semantic function I:

$$\begin{aligned} I[\![\text{var } x]\!] &= \lambda e. \, . \, e(x) \\ I[\![x \text{and} y]\!] &= \lambda e. \, . \, I[\![x]\!] \; e \wedge I[\![y]\!] \; e \\ I[\![x or y]\!] &= \lambda e. \, . \, I[\![x]\!] \; e \vee I[\![y]\!] \; e \end{aligned}$$

Embedding OCL into Isabelle/HOL

The Challenges of OCL

1. OCL semantics evaluation-oriented (undefinedness $\perp$, strict evaluation)
2. large library in operational style
3. logics three-valued (Kleene-Logics) (i.e. $a \wedge true = a$ and $true \wedge a = a$ for $a \in \{true, false, \perp_{\mathscr{L}}\}$)
4. reasoning over UML path expressions

Defining the Library

- The OCL standard describes:

$$I(\cup)(X,Y) \equiv \begin{cases} X \cup Y & \text{if } X \neq \bot_{\mathscr{L}} \text{ and } Y \neq \bot_{\mathscr{L}} \\ \bot_{\mathscr{L}} & \text{otherwise} \end{cases}$$

- In Isabelle we define:

```
constdefs
 union :: " [('a,('b::bot) Set) VAL, ('a,'b Set) VAL]
                                  ⇒ ('a,'b Set) VAL"
                                          ("_ ->union'( _ ')" [66,65]65)
  "union ≡ lift2 ( strictify (λX. strictify (λY. Abs_Set
            (⌊ Lifting.drop (Rep_Set X) ∪ Lifting.drop (Rep_Set Y)⌋)
     )))"
```

But is this faithful to the standard? (1/2)

Let's consider the *not*:

$$
\begin{aligned}
\text{Not X} &= \text{lift1 not' X} \\
&= \text{lift1} \ \lfloor_\rfloor \circ \ \text{strictify} \ (\neg \circ \lceil_\rceil) \ \text{X} \\
&= \lambda \text{St.} \ \text{if DEF(X St)} \\
&\qquad\qquad \text{then} \ \lfloor \neg \lceil \text{X St} \rceil \rfloor \\
&\qquad\qquad \text{else} \ \bot
\end{aligned}
$$

But is this faithful to the standard? (1/2)

Let's consider the not:

$$\begin{aligned}\mathsf{Not\ X} &= \mathsf{lift1\ \ not'\ \ X}\\ &= \mathsf{lift1}\ \ \lfloor_\rfloor \circ\ \ \mathsf{strictify}\ \ (\neg \circ \lceil_\rceil)\ \mathsf{X}\\ &= \lambda \mathsf{St.}\ \ \mathsf{if\ \ DEF(X\ St)}\\ &\qquad\qquad \mathsf{then}\ \lfloor\ \neg \lceil \mathsf{X\ St}\rceil\ \rfloor\\ &\qquad\qquad \mathsf{else}\ \ \bot\end{aligned}$$

By introducing the usual semantic function Sem (which happens to be the identity in a shallow embedding), we get:

$$\begin{aligned}\mathsf{Sem}[\![\mathsf{Not\ X}]\!]\ \mathsf{St} &= \mathsf{if\ DEF(Sem}[\![\mathsf{X}]\!]\ \mathsf{St)}\\ &\qquad\qquad \mathsf{then}\ \lfloor\ \neg \lceil \mathsf{Sem}[\![\mathsf{X}]\!]\ \mathsf{St}\rceil\ \rfloor\\ &\qquad\qquad \mathsf{else}\ \ \bot\end{aligned}$$

But is this faithful to the standard? (2/2)

- Which looks already like the definition in the standard

$$I(\neg)(X) \equiv \begin{cases} \neg X & \text{if } X \neq \bot_{\mathscr{L}} \\ \bot_{\mathscr{L}} & \text{otherwise} \end{cases}$$

- Further, we prove (beeing conservative) all the usual laws (idempotency, associativity, . . .) for all defined operators.
- But this is a lot of work . . .

But is this faithful to the standard? (2/2)

- Which looks already like the definition in the standard

$$I(\neg)(X) \equiv \begin{cases} \neg X & \text{if } X \neq \bot_{\mathscr{L}} \\ \bot_{\mathscr{L}} & \text{otherwise} \end{cases}$$

- Further, we prove (beeing conservative) all the usual laws (idempotency, associativity, . . .) for all defined operators.
- But this is a lot of work . . . Let's try to automate [BW03] this

Library Construction

Problem: Large library in operational style

Library Construction

Problem: Large library in operational style

Answer: Combinator Style Semantics building a Theory Morphism

Library Construction

Problem: Large library in operational style

Answer: Combinator Style Semantics building a Theory Morphism
Representing Phases of the Translation
by combinators on

1. types (type constructors)
2. terms (higher-order functions)

once and for all!

Library Construction

Problem: Large library in operational style

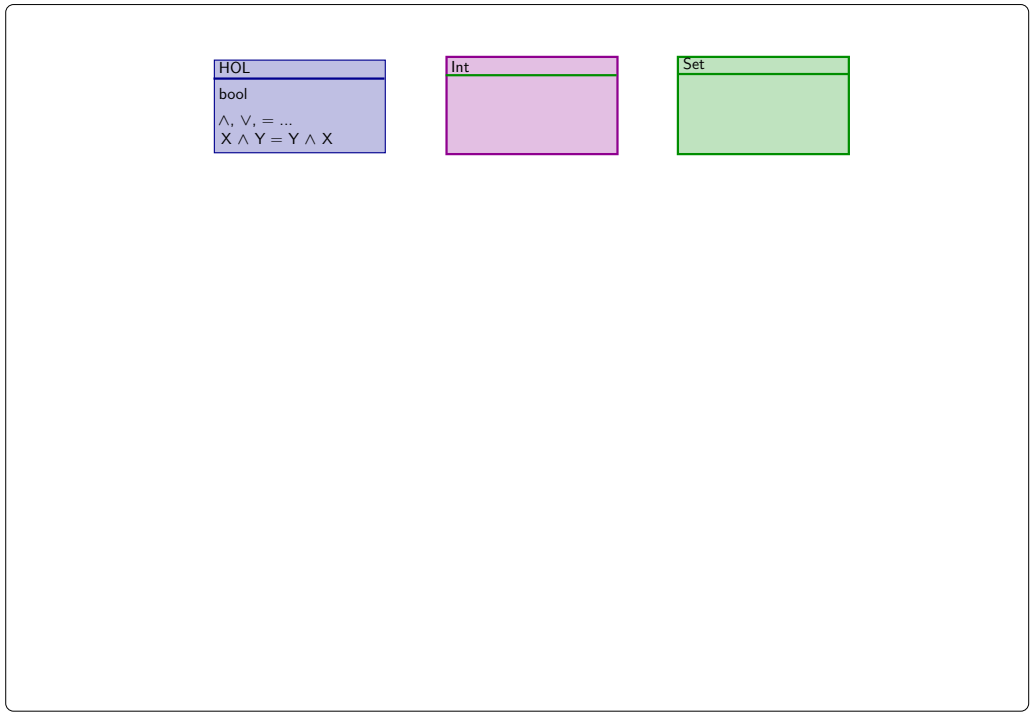
Answer: Combinator Style Semantics building a Theory Morphism
Representing Phases of the Translation
by combinators on

1. types (type constructors)
2. terms (higher-order functions)

once and for all!
Develop mechanical support for "Lifting Theorems".

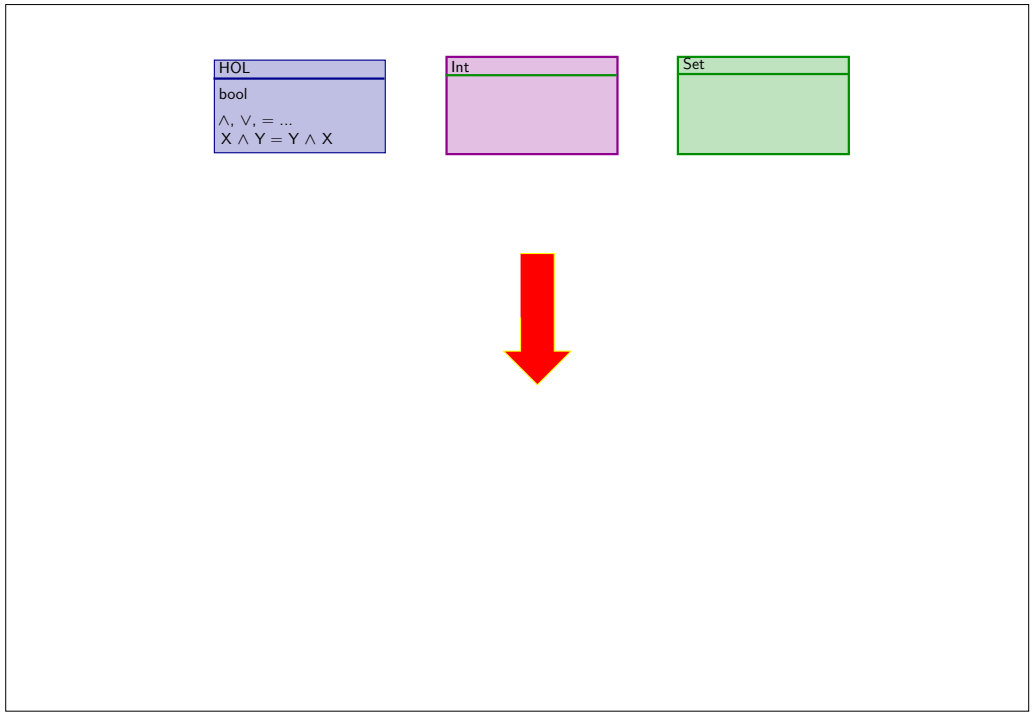
Observation:

90 percent of the embedding is a canonical theory morphism!



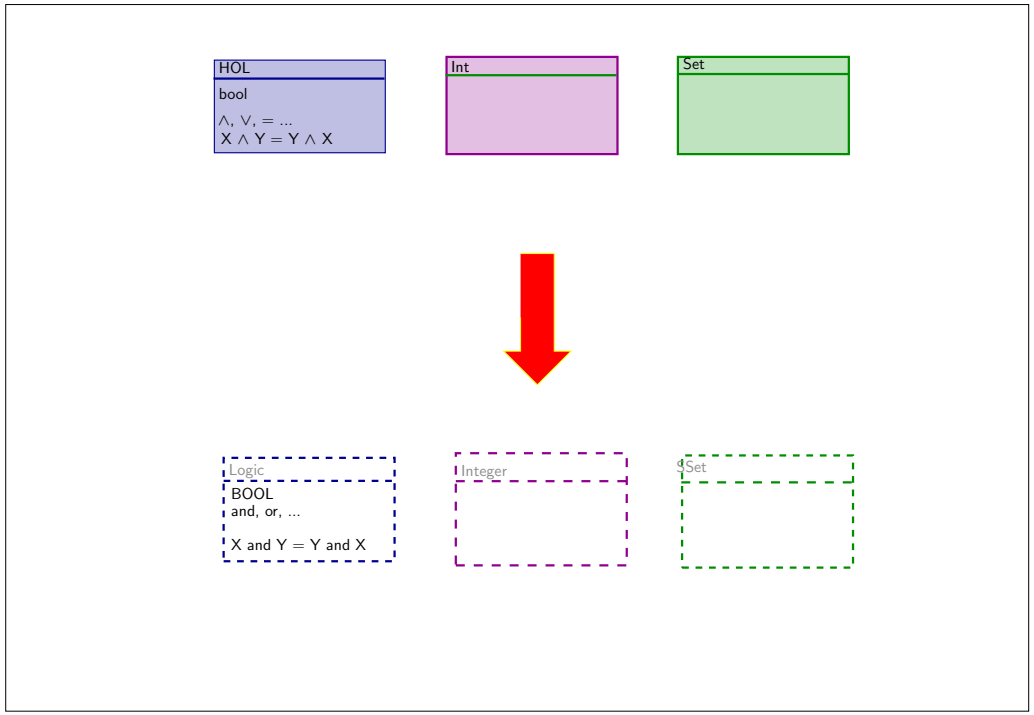
Observation:

90 percent of the embedding is a canonical theory morphism!



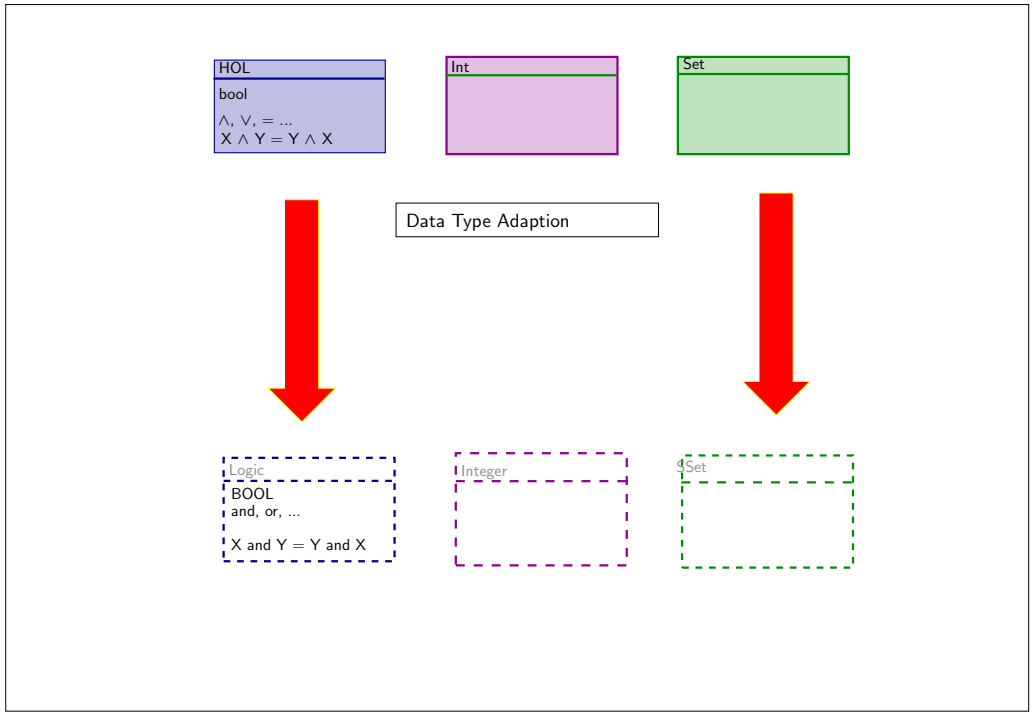
Observation:

90 percent of the embedding is a canonical theory morphism!



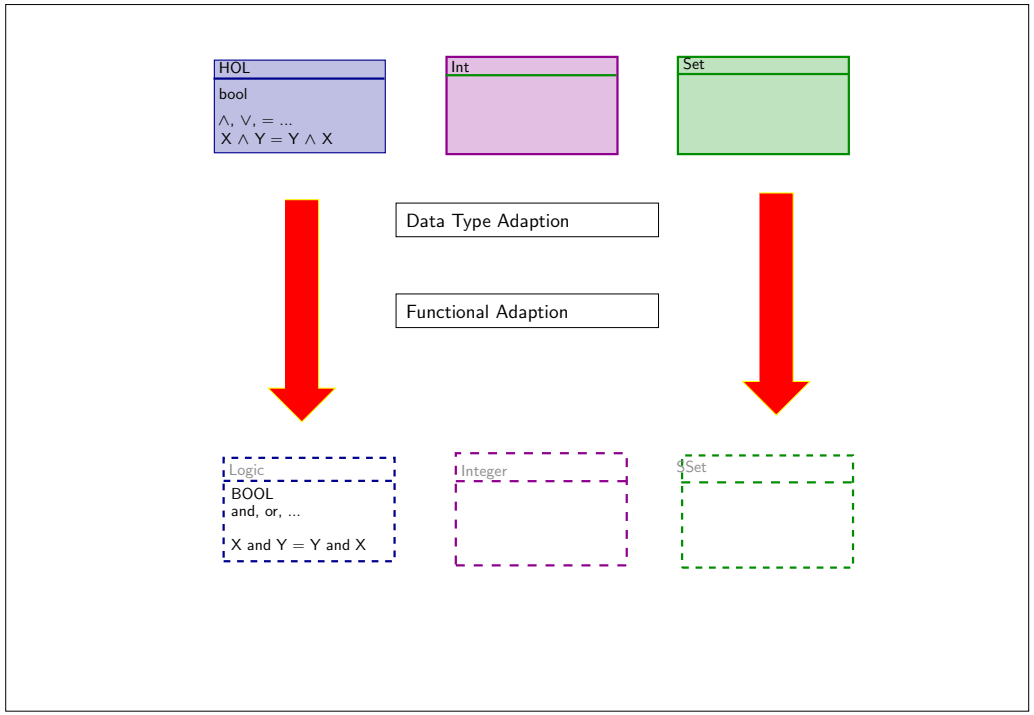
Overall Idea:

Organize the morphism into phases represented by "semantic combinators"!



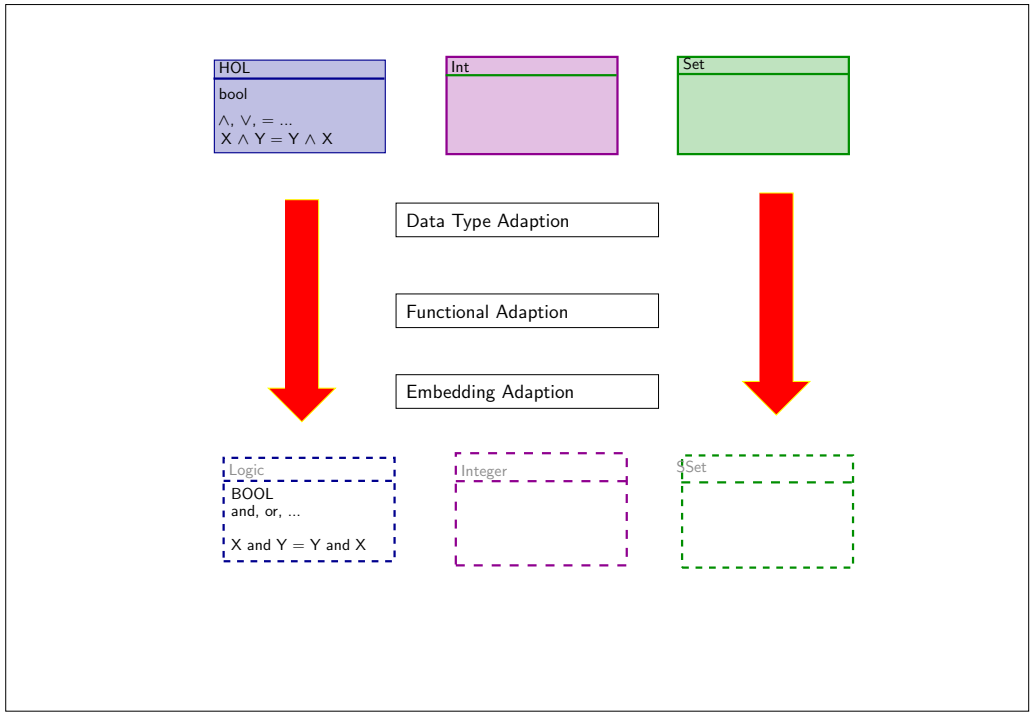
Overall Idea:

Organize the morphism into phases represented by "semantic combinators"!



Overall Idea:

Organize the morphism into phases represented by "semantic combinators"!



Advantages of the Combinator Representation

- generic theorems for strictness and definedness
  lead to tactics that construct specialized theorems
  for each operation in the theory morphism, e.g.: Example
  $\mathsf{DEF}(\,\mathsf{strictify}\;\; \mathsf{f}\;\; \mathsf{X}) \Longrightarrow\; \mathsf{DEF}\;\mathsf{f} \wedge \mathsf{DEF}\;\mathsf{X}$

- generic "lifting theorems" for commutativity . . .
  allowing to "lift" these properties automatically . . .
  Example

$$
\begin{aligned}
&(\bigwedge \mathsf{x}\;\mathsf{y}.\;\; \mathsf{f}\;\;\mathsf{x}\;\;\mathsf{y} = \;\mathsf{f}\;\;\mathsf{y}\;\;\mathsf{x}) \Longrightarrow \\
&\quad\;\; (\;\mathsf{strictify}\;\;\; (\lambda \mathsf{x}.\;\;\; \mathsf{strictify}\;\;(\mathsf{f}\;\;\mathsf{x})))\;\;\mathsf{X}\;\;\mathsf{Y} \\
&\;\; = (\;\mathsf{strictify}\;\;\; (\lambda \mathsf{x}.\;\;\; \mathsf{strictify}\;\;(\mathsf{f}\;\;\mathsf{x}))\;\;\mathsf{Y}\;\;\mathsf{X})
\end{aligned}
$$

Do we honour the standard?

- The "high-level meaning" of the operators is given as pre-/post-condition pairs, e.g.:

```
context: Collection.count(object:T):integer
post: result = self→iterate(elem;
                            acc: integer = 0
                            | if elem = object then acc
```

- These can be proven in HOL-OCL:

```
lemma "((self::('a, 'b::bot Set)VAL)->count((obj::('a, 'b)VAL))) =
                    (if self->includes(obj)
                     then 1 else 0 endif)"
```

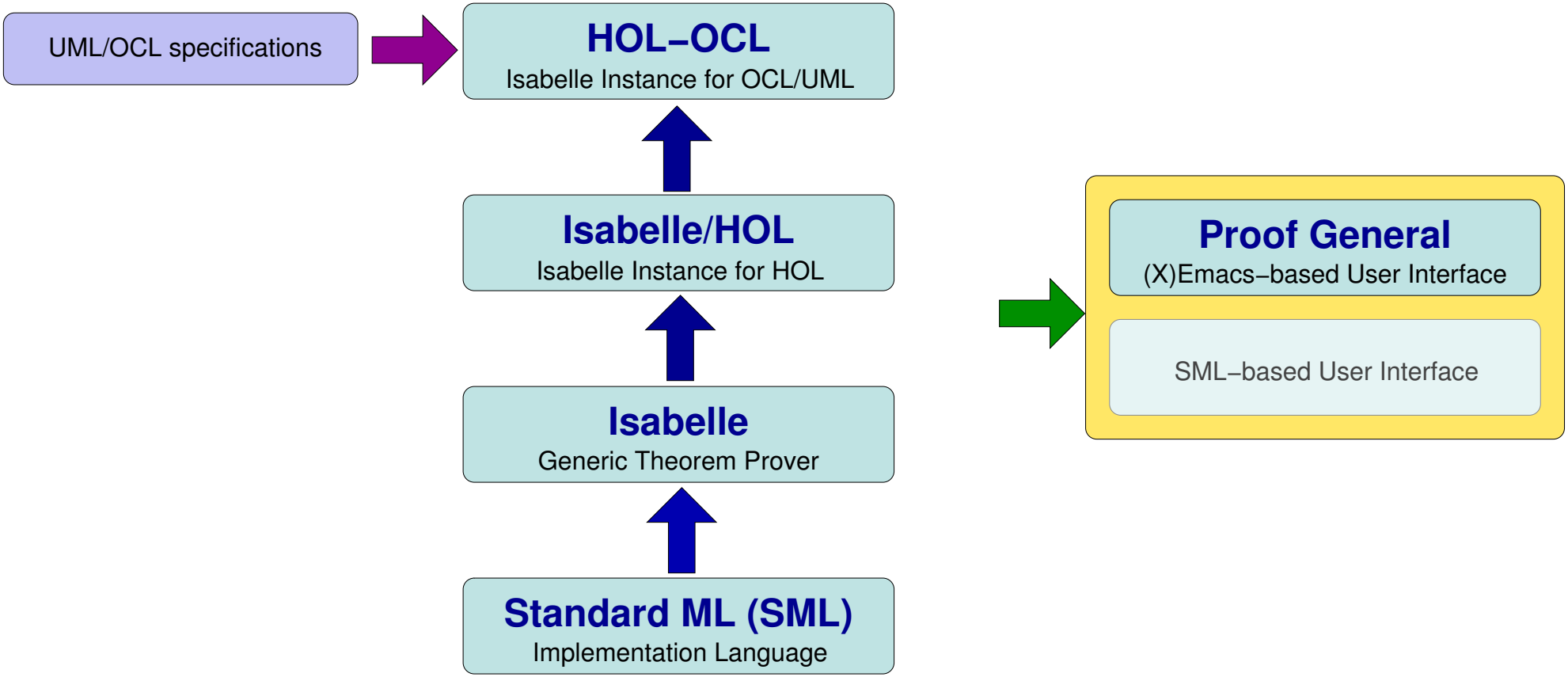
```
lemma "((self::('a, 'b::bot Set)VAL)->count((obj::('a, 'b)VAL))) =
                                  ( if  self->includes(obj)
                                   then 1 else 0 endif )"
apply ( rule ext )
apply (simp add:  count_def  includes_def  ocl_if_def  weak_eq_def
                  OCL_Integer. Zero_ocl_int_def
                  OCL_Integer. One_ocl_int_def )
apply (simp only:  lift0_def  lift1_def  lift2_def  lift3_def
                   strictify_def  o_def  not_def  oclIsDefined_def  T
                  RUE_def)
apply ( simp_all  (no_asm_use) add: UU_fun_def DEF_def  strictify_def
                 split  add:  split_if  up. split )
done
```

Summing up: The OCL-Level

- We have seen:
  - Generic Theorems for handling undefindness, lifting, etc.
  - Generic Support for Calculi
  - Automated "Lifting" of theorems from the HOL level to the OCL level
- Overall, HOL-OCL provides:
  - a consistend OCL semancics build as an conservative (shallow) embedding into Isabelle/HOL
  - proof support for OCL (several calculi)

Conclusion

The overall picture



Conclusion

- formal semantics of UML/OCL 2.0

Conclusion

- formal semantics of UML/OCL 2.0
- layered, "combinator-style" representation

Conclusion

- formal semantics of UML/OCL 2.0
- layered, "combinator-style" representation
- derived library

Conclusion

- formal semantics of UML/OCL 2.0
- layered, "combinator-style" representation
- derived library
- derived 3-valued logical calculi
  inlcuding machine-support

Conclusion

- formal semantics of UML/OCL 2.0
- layered, "combinator-style" representation
- derived library
- derived 3-valued logical calculi
  inlcuding machine-support
- extensible, typed object-state
  enabling subtyping on dynamic types

HOL-OCL: A Shallow Embedding of OCL into HOL

The HOL-OCL system [BW02a, BW02b]:

- is build on top of Isabelle/HOL.
- is a shallow embedding of OCL into HOL.
- provides a consistent (machine checked) OCL semantics.
- allows the examination of OCL features.
- builds the basis for OCL tool development.
- follows OCL 2.0
- over 2000 theorems (language properties) proven.

The Technical Design of HOL-OCL

- Reuseability:
  - Reuse old proofs for class diagrams constructed via inheritance introduction of new classes.
  - Extendible semantics approach.
- Representing semantics structurally:
  - Organize semantic definitions by certain combinators capturing the semantical essence (e.g. lifting and strictness).
  - Automatically construct theorems out of uniform definitions.

Higher-Order Logic Applications: Other

Achim D. Brucker

Isabelle is not alone

there are many successful theorem provers, e.g.:

- Coq:

  http://coq.inria.fr/

- HOL4:

  http://hol.sourceforge.net/

- ACL2:

  http://www.cs.utexas.edu/users/moore/acl2/

- PVS:

  http://pvs.csl.sri.com/

- . . .

Now let's look at some applications

Interactive theorem proving is applied in several areas, e.g.:

- Environments for Formal Specifications
- Hardware Verification
- Protocol verification
- . . .

This selection is neither objective nor complete!

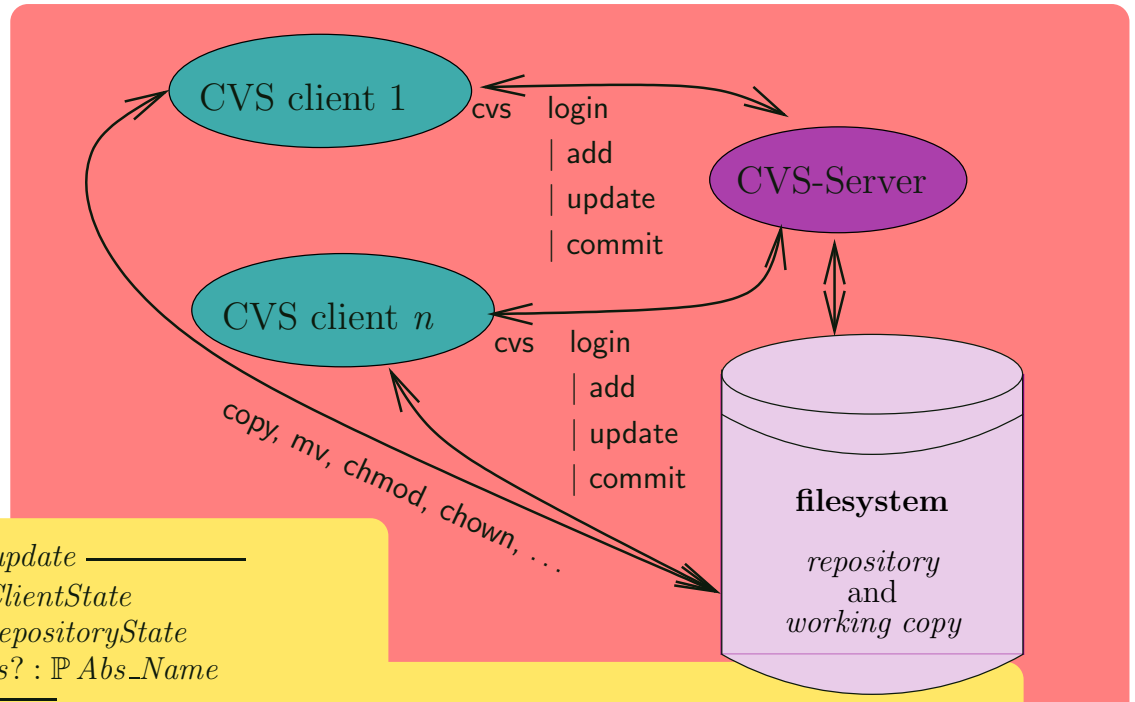
Proof Environments for Formal Specification Languages

Isabelle/HOL-Z

HOL-Z [BRW03b, hol] is a proof environment for Z and HOL specifications.

- Z [iso00] is an data-oriented modeling language.
- HOL-Z is a shallow embedding of Z into Isabelle/HOL.
- HOL-Z allows to reason formally over Z specifications.
- As an extensions, one can combine HOL and Z.
- HOL-Z was successfully used for analyzing large security architectures:
  - Access Control for CVS [BW05]
  - A architecture for administering digital signatures [BKTW05]

HOL-Z: Applications 1/2

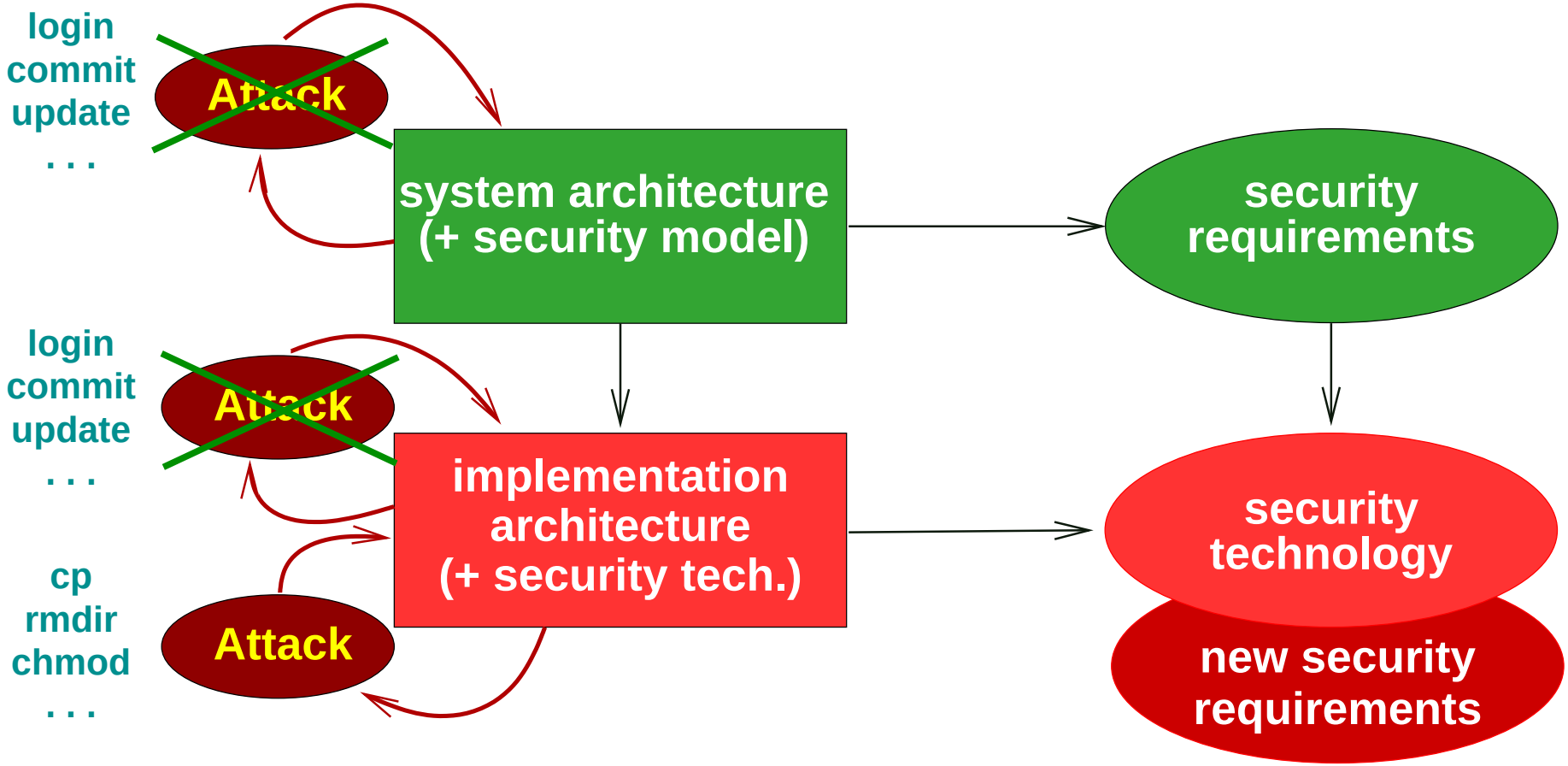


update

$\Delta ClientState$
$\Xi RepositoryState$
$files? : \mathbb{P}\, Abs_Name$

$wc' = wc \oplus \{n : wfiles \cap files? \mid n \in \mathsf{dom}\, rep \wedge n \in \mathsf{dom}\, wc_uidtab$
$\quad \wedge\ (wc_uidtab(n), abs_passwd(wc_uidtab\ n))\ \mathsf{is_valid_in}\ rep\} \lhd rep)$
$wc_uidtab' = wc_uidtab \cup \{n : wfiles \cap files? \mid n \in \mathsf{dom}\, rep$
$\quad \wedge\ n \notin \mathsf{dom}\, wc_uidtab \bullet n \mapsto choose_valid_rolename(rep_permtab, n)\}$
$abs_passwd' = abs_passwd \wedge wfiles' = wfiles$

HOL-Z: Applications 2/2



Hardware Verification

The Pentium FDIV Bug

- What is the result of

$$4195835 - 3145627 \cdot \frac{4195835}{3145627} = \left\{ \right.$$

The Pentium FDIV Bug

- What is the result of

$$4195835 - 3145627 \cdot \frac{4195835}{3145627} = \begin{cases} 0 & \text{as expected, or} \end{cases}$$

The Pentium FDIV Bug

- What is the result of

$$4195835 - 3145627 \cdot \frac{4195835}{3145627} = \begin{cases} 0 & \text{as expected, or} \\ 256 & \text{on a Pentium} \end{cases} \quad ?$$

The Pentium FDIV Bug

- What is the result of

$$4195835 - 3145627 \cdot \frac{4195835}{3145627} = \begin{cases} 0 & \text{as expected, or} \\ 256 & \text{on a Pentium} \end{cases} \quad ?$$

- The first series of Pentium processors had several wrong entries in a lookup table used by the division-algorithm.
- The cost for replacing the defect chips did cost Intel over 475 million USD in 4th quarter of 1994
- The showpiece of chip-verification, Binary Decision Diagrams (BDDs), don't work well for division [SBW98].

Verifying Floating Point arithmetic

"Motivated" by the Pentium Bug [Ede97, Pra95], theorem proving became widely used in the chip industry,e.g.:

at Intel: John Harrision verified several floating point algorithms [Har98, Har99, Har00], among them the division of the IA64 architecture using HOL Light.

at AMD: Since the AMD K5 (1995) all important arithmetic algorithms (and much more) are verified using ACL2, see e.g. [Obe97, Rus98].

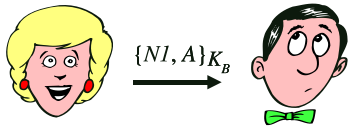
Today, theorem proving (combined with BDD-based model checking) is an everyday tool used in hardware verification..

Isabelle/HOL: Protocol Verification

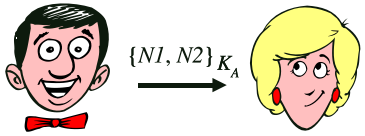
Isabelle/HOL: Protcol Verification

Needham-Schroeder Public Key Protocol (1970):

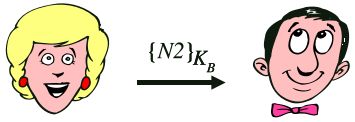
1. $A \rightarrow B: \ \{NA, A\}_{K_B}$
2. $B \rightarrow A: \ \{NA, NB\}_{K_A}$
3. $A \rightarrow B: \ \{NB\}_{K_B}$



"This is Alice and I have chosen a nonce NA."



"Here is your nonce NA. Since I could read it, I must be Bob. I've also a challenge NB for you."

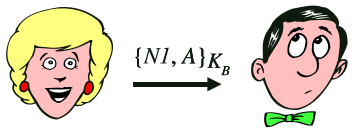


"You sent me NB. Since only Alice can read this and I sent it back, you must be Alice."

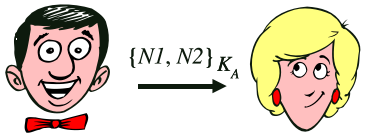
Isabelle/HOL: Protcol Verification

Needham-Schroeder Public Key Protocol (1970):

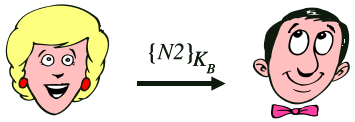
1. $A \rightarrow B: \ \{NA, A\}_{K_B}$
2. $B \rightarrow A: \ \{NA, NB\}_{K_A}$
3. $A \rightarrow B: \ \{NB\}_{K_B}$



"This is Alice and I have chosen a nonce NA."



"Here is your nonce NA. Since I could read it, I must be Bob. I've also a challenge NB for you."



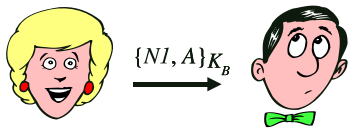
"You sent me NB. Since only Alice can read this and I sent it back, you must be Alice."

Protocols are typically small and convincing...

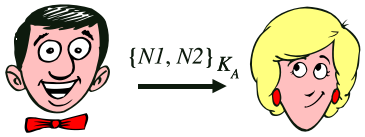
Isabelle/HOL: Protcol Verification

Needham-Schroeder Public Key Protocol (1970):

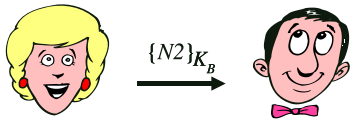
1. $A \rightarrow B: \ \{NA, A\}_{K_B}$
2. $B \rightarrow A: \ \{NA, NB\}_{K_A}$
3. $A \rightarrow B: \ \{NB\}_{K_B}$



"This is Alice and I have chosen a nonce NA."



"Here is your nonce NA. Since I could read it, I must be Bob. I've also a challenge NB for you."



"You sent me NB. Since only Alice can read this and I sent it back, you must be Alice."

Protocols are typically small and convincing...

and often wrong!

Analzying Security Protocols

- Larry Paulson has developed a new approach to the verification of cryptographic protocols. The operational semantics of all agents in the network (including an active intruder) is modeled using a series of inductive definitions [Pau98].
- Christoph Sprenger formalized a cryptographic library and a generic proof environment for security protocols using Isabelle/HOL [Spr]. This is the first formal verification approach of security protocols that also considers "real world" cryptographic operators.

Last Words

Pointers for Further Studies

- The Archive of Formal Proofs: http://afp.sf.net
- Isabelle's Projects Page: http://www.cl.cam.ac.uk/Research/HVG/Isabelle/projects.html
- The Formal Methods Page: http://vl.fmnet.info/
- MIZAR: http://mizar.org/JFM/

Pointers for Further Studies

- The Archive of Formal Proofs: http://afp.sf.net
- Isabelle's Projects Page: http://www.cl.cam.ac.uk/Research/HVG/Isabelle/projects.html
- The Formal Methods Page: http://vl.fmnet.info/
- MIZAR: http://mizar.org/JFM/

Now find your own killer application
for theorem proving and . . .

. . . apply what you have learned!

. . . apply what you have learned!

We always offer Isabelle related thesis projects. You can directly participate in the further development of

- HOL-OCL,
- HOL-Z,
- formal test-case generation,

or develop your own case-study, e.g. formalize red-black trees.

Just ask or look at our web page:

`http://www.infsec.ethz.ch/education/projects`!

References

[Abr96] J.-R. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.

[Acz77] Peter Aczel. *Handbook of Mathematical Logic*, chapter An Introduction to Inductive Definitions, pages 739–782. North-Holland, 1977.

[And86] Peter B. Andrews. *An Introduction to Mathematical Logic and Type Theory: To Truth Through Proofs*. Academic Press, 1986.

[Ari] Aristotle. *Analytica priora I*, chapter 4.

[A.W97] A.W.Roscoe. *The Theory and Pactice of Concurrency*. Prentice Hall, 1997.

[BKTW05] David Basin, Hironobu Kuruma, Kazuo Takaragi, and Burkhart Wolff. Specifying and verifying hysteresis signature system with HOL-Z. Technical Report 471, Computer Security Group, ETH Zürich, 1 2005.

[BN98] Franz Baader and Tobias Nipkow. *Term Rewriting and* All That. Cambridge University Press, 1998.

[BRW03a] Achim D. Brucker, Frank Rittinger, and Burkhart Wolff. Hol-z 2.0: A proof environment for z-specifications. *Journal of Universal Computer Science*, 9(2):152–172, February 2003.

[BRW03b] Achim D. Brucker, Frank Rittinger, and Burkhart Wolff. HOL-Z 2.0: A proof environment for Z-specifications. *Journal of Universal Computer Science*, 9(2):152–172, February 2003.

[BW02a] Achim D. Brucker and Burkhart Wolff. HOL-OCL: Experiences, consequences and design choices. In Jean-Marc Jézéquel, Heinrich Hussmann, and Stephen Cook, editors, *UML 2002: Model Engineering, Concepts and Tools*, number 2460, pages 196–211. Springer-Verlag, Dresden, 2002.

[BW02b] Achim D. Brucker and Burkhart Wolff. A proposal for a formal OCL semantics in Isabelle/HOL. In César Muñoz, Sophiène Tahar, and Víctor

Carreño, editors, *Theorem Proving in Higher Order Logics*, number 2410, pages 99–114. Springer-Verlag, Hampton, VA, USA, 2002.

[BW03] Achim D. Brucker and Burkhart Wolff. Using theory morphisms for implementing formal methods tools. In Herman Geuvers and Freek Wiedijk, editors, *Types for Proof and Programs*, number 2646, pages 59–77. Springer-Verlag, Nijmegen, 2003.

[BW05] Achim D. Brucker and Burkhart Wolff. A verification approach for applied system security. *International Journal on Software Tools for Technology Transfer (STTT)*, 2005. to appear.

[CDTK86] J. Cl ´ement, J. Despeyroux, T.Despeyroux, and G. Kahn. A simple applicative language: mini-ml. In *Proc. of the 1986 ACM Conference on Lisp and Functional Programming*. ACM, 1986.

[Chu40] Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[Ede97] Alan Edelman. The mathematics of the pentium division bug. *SIAM*, 39(1):54–67, 1997.

[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.

[Fre93] Gottlob Frege. *Grundgesetze der Arithmetik*, volume I. Verlag Hermann Pohle, 1893. Translated in part in [Fur64].

[Fre03] Gottlob Frege. *Grundgesetze der Arithmetik*, volume II. Verlag Hermann Pohle, 1903. Translated in part in [Fur64].

[Fur64] Montgomery Furth. *The Basic Laws of Arithmetic*. Berkeley: University of California Press, 1964. Translation of [Fre03].

[Gen35] Gerhard Gentzen. Untersuchungen über das logische Schliessen. *Mathe-*

matische 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.

[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.

[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.

[hol] http://abacus.informatik.uni-freiburg.de/holz/.

[Höl90] Steffen Hölldobler. Conditional equational theories and complete sets of transformations. *Theoretical Computer Science*, 75(1&2):85–110, 1990.

[iso00] Formal Specification – Z Notation – Syntax, Type and Semantics, June 2000. Consensus Working Draft 2.6.

[Klo93] Jan Willem Klop. *Handbook of Logic in Computer Science*, chapter "Term Rewriting Systems". Oxford: Clarendon Press, 1993.

[LP81] Harry R. Lewis and Christos H. Papadimitriou. *Elements of the Theory of Computation*. Prentice-Hall, 1981.

[MNOS99] Olaf Müller, Tobias Nipkow, David von Oheimb, and Oskar Slotosch. HOLCF = HOL + LCF. *Journal of Functional Programming*, 9:191–223, 1999.

[Nip93] Tobias Nipkow. *Logical Environments*, chapter Order-Sorted Polymorphism in Isabelle, pages 164–188. Cambridge University Press, 1993.

[Nip02] Tobias Nipkow. Hoare logics in Isabelle/HOL. In H. Schwichtenberg and R. Steinbrüggen, editors, *Proof and System-Reliability*, pages 341–367. Kluwer, 2002.

[NOP00] Tobias Nipkow, David von Oheimb, and Cornelia Pusch. μJava: Embedding a programming language in a theorem prover. In F.L. Bauer and

R. Steinbrüggen, editors, *Foundations of Secure Computation. Proc. Int. Summer School Marktoberdorf 1999*, pages 117–144. IOS Press, 2000.

[NP95] Tobias Nipkow and Christian Prehofer. Type reconstruction for type classes. *Journal of Functional Programming*, 5(2):201–224, 1995.

[Obe97] Stuart F. Oberman. Floating point division and square root algorithms and implementation in the AMD-K7^{TM} microprocessor. Technical report, Advanced Micro Devices, Sunyvale, CA 94088, March 1997.

[omg01] OMG Unified Modeling Language Specification, September 2001.

[omg03] Uml 2.0 ocl specification, October 2003.

[Pau96] Lawrence C. Paulson. *ML for the Working Programmer*. Cambridge University Press, 1996.

[Pau98] Lawrence C. Paulson. The inductive approach to verifying cryptographic protocols. *J. Comput. Secur.*, 6(1-2):85–128, 1998.

[Pau03] Lawrence C. Paulson. *The Isabelle Reference Manual*. Computer Laboratory, University of Cambridge, March 2003.

[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.

[Pra95] V.R. Pratt. Anatomy of the pentium bug. In *TAPSOFT'95*, volume 915 of *LNCS*, pages 97–107, Aarhus, Denmark, 1995. Springer-Verlag.

[Rus98] David M. Russinoff. A mechanically checked proof of IEEE compliance of the floating point multiplication, division and square root algorithms of the AMD-K7^{TM} processor. *LMS JCM*, 1:148–200, December 1998.

[RW04] Nicole Rauch and Burkhart Wolff. Formalizing java's two's-complement integral type in isabelle/hol. Technical Report 458, ETH Zürich, 11 2004.

[SBW98] C. Scholl, B. Becker, and T.M. Weis. Word-level decision diagrams, wlcds and division. pages 672–677, 1998.

[Spi92] J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall International Series in Computer Science, 2nd edition, 1992.

[Spr] http://www.zisc.ethz.ch/research/securityprotocolproofs.

[Sza69] M. E. Szabo. *The Collected Papers of Gerhard Gentzen*. North-Holland, 1969.

[Tho95a] Della Thompson, editor. *The Concise Oxford Dictionary*. Clarendon Press, 1995.

[Tho95b] Simon Thompson. *Miranda: The Craft of Functional Programming*. Addison-Wesley, 1995.

[Tho99] Simon Thompson. *Haskell: The Craft of Functional Programming*. Addison-Wesley, 1999. Second Edition.

[TW97] H. Tej and B. Wolff. A corrected failure-divergence model for csp in isabelle/hol. In J. Fitzgerald, C.B. Jones, and P. Lucas, editors, *Proceedings of the FME 97 — Industrial Applications and Strengthened Foundations of Formal Methods*, LNCS 1313, pages 318–337. Springer Verlag, 1997.

[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.

[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.

[WD96] Jim Woodcock and Jim Davies. *Using Z: Specification, Refinement, and Proof*. Prentice Hall International Series in Computer Science. Prentice Hall, 1996.

[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.