Specification and Verification in Higher Order Logic

Prof. Dr. K. Madlener

20. Juli 2011

Chapter 0

Organisation, Overview

Organisation

Contact

- Klaus Madlener
- Patrick Michel
- Christoph Feller
- **http://www-madlener.informatik.uni-kl.de/teaching/ss2011/**

Dates, Time, and Location

- 3C + 3R (8 ECTS-LP)
- Monday, 11:45-13:15, room 48-462
- Wednesday, 11:45-13:15, room 48-462 or room 32-411
- Thursday, 11:45-13:15, room 48-462

Organisation

Course Webpage

- **http://www-madlener.informatik.uni-kl.de/teaching/ss2011/svhol/**

Literature

Organisation (cont.)

- L. C. Paulson. *ML for the Working Programmer.* Cambridge University Press, 1996.
- R. Harper. *Programming in Standard ML.* Available at http://www.cs.cmu.edu/ rwh/smlbook/book.pdf. Carnegie Mellon University, 2009.
- T. Nipkow, L. C. Paulson and M. Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic.* Springer LNCS 2283, 2002
- Prof. Basin, Dr. Brucker, Dr. Smaus, Prof. Wolff *Material of course CSMR* - http://www.infsec.ethz.ch/education/permanent/csmr/slides

Organisation (cont.)

Acknowledgements

- to Dr. Jens Brandt who designed most of the slides
- Prof. Dr. Arnd Poetzsch-Heffter for providing his course material
- Prof. Basin, Dr. Brucker, Dr. Smaus, Prof. Wolff, and the MMISS-project for the slides on CSMR
- Prof. Nipkow for the slides on Isabelle/HOL.
- to the Isabelle/HOL community

Overview

Course Outline

- Introduction
- Concepts of functional programming
- Higher-order logic
- Verification in Isabelle/HOL (and other theorem provers)
- Verification of algorithms: A case study
- Modeling and verification of finite software systems: A case study
- Specification of programming languages
- Verification of a Hoare logics
- Beyond interactive theorem proving

Overall structure

1. Introduction
2. Functional specification and programming
3. Language and semantical aspects of higher-order logic
4. Proof system for higher-order logic
5. Sets, functions, relations, and fixpoints
6. Verifying functions
7. Inductively defined sets
8. Specification of programming language semantics
9. Program verification and programming logic

Chapter 1: Introduction

1. Terminology: Specification, verification, logic
2. Language: Syntax and semantics
3. Proof systems
   3.1 Hilbert style proof systems
   3.2 Proof system for natural deduction

Chapter 2: Functional programming and specification

1. Functional programming in ML
2. A simple theorem prover: Structure and unification
3. Functional specification in isabelle/HOL

» `slides_02: 1-65`
» `slides_02: 77-101`
» `Chapter 2 and 3 of Isabelle/HOL Tutorial`

Chapter 3: Language and semantical aspects of HOL

1. Introduction to higher-order logic
2. Foundation of higher-order logic
3. Conservative extension of theories

Chapter 4: Proof system for HOL

1. Formulas, sequents, and rules revisited
2. Application of rules
3. Fundamental methods of Isabelle/HOL
4. An overview of theory Main
   4.1 The structure of theory Main
   4.2 Set construction in Isabelle/HOL
   4.3 Natural numbers in Isabelle/HOL

Chapter 4: Proof system for HOL (cont.)

5. Rewriting and simplification
6. Case analysis and structural induction
7. Proof automation
8. More proof methods

» `slides of Sessions 2, 3.1, 3.2, and 4 & 5 by T. Nipkow`
» `Chapter 5 of Isabelle/HOL Tutorial til page 99`

Chapter 5: Sets, functions, relations, and fixpoints

1. Sets
2. Functions
3. Relations
4. Well-founded relations
5. Fixpoints

» `Chapter 6 of Isabelle/HOL Tutorial til page 118`

Chapter 6: Verifying functions

1. Conceptual aspects
2. Case study: Gcd
3. Case study: Quicksort – Shallow embedding of algorithms

» `theories for Gcd and Quicksort`

Chapter 7: Inductively defined sets

1. Defining sets inductively
2. Specification of transitions systems
   - 2.1 Transition systems
   - 2.2 Modeling: Case study Elevator
   - 2.3 Reasoning about finite transition systems

» Section 7.1 of Isabelle/HOL Tutorial
» slides of Sessions 6.1 T. Nipkow
» theory for Elevator

Chapter 8:
Specification of programming language semantics

1. Introduction to programming language semantics
2. Techniques to express semantics
   2.1 Natural semantics / big step semantics
   2.2 Structured operational semantics / small step semantics
   2.3 Denotational semantics
3. Formalizing semantics in HOL

» `slides about operational semantics by P. MÃ¼ller`

» `theory for while-language`

Chapter 9:
Program verification and programming logic

1. Hoare logic
2. Program verification based on language semantics
3. Program verification with Hoare logic
4. Soundness of Hoare logic

» `theory for while-language`
» `theory for Hoare logic`

Chapter 1

Introduction

Overview

Motivation

- Specifications: Models and properties ⇝ Spec-formalisms
- How do we express/specify facts? ⇝ Languages
- What is a proof? What is a formal proof? ⇝ Logical calculus
- How do we prove a specified fact? ⇝ Proof search
- Why formal? What is the role of a theorem prover? ⇝ Tools

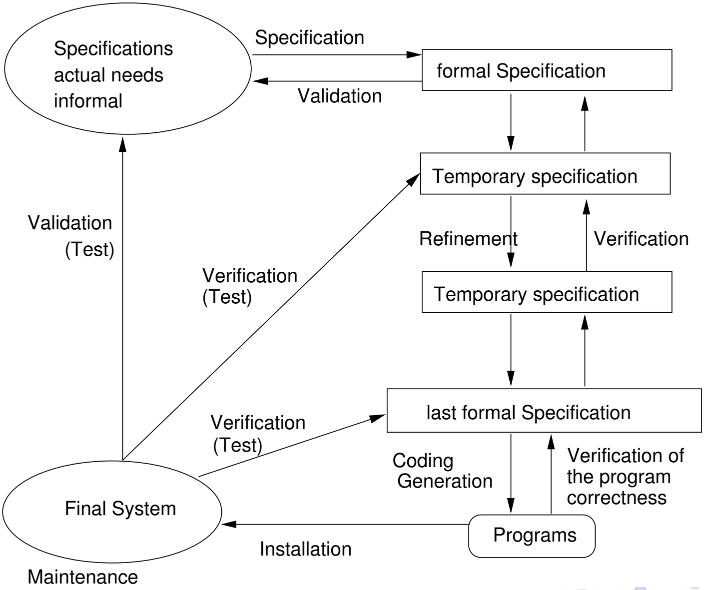
Goals

- role of formal specifications
- recapitulate logic
- introduce/review basic concepts

Role of formal Specifications

- Software and hardware systems must accomplish well defined tasks (**requirements**).
- Software Engineering has as goal
  - Definition of criteria for the evaluation of SW-Systems
  - Methods and techniques for the development of SW-Systems, that accomplish such criteria
  - Characterization of SW-Systems
  - Development processes for SW-Systems
  - Measures and Supporting Tools
- Simplified view of a SD-Process:
  Definition of a sequence of actions and descriptions for the SW-System to be developed. Process- and Product-Models

  Goal: The group of documents that includes an executable program.


Specifications
actual needs
informal
Specification
Validation
formal Specification
Temporary specification
Refinement
Verification
Temporary specification
Validation
(Test)
Verification
(Test)
last formal Specification
Verification
(Test)
Coding
Generation
Verification of
the program
correctness
Final System
Programs
Installation
Maintenance

Comment

- First Specification: Global Specification
  Fundament for the Development
  "Contract or Agreement" between Developers and Client
- Intermediate (partial) specifications:
  Base of the Communication between Developers.
- Programs: Final products.

Development paradigms

- Structured Programming
- Design + Program
- Transformation Methods
- ...

Properties of Specifications

Consistency Completeness

- Validation of the global specification regarding the requirements.
- Verification of intermediate specifications regarding the previous one.
- Verification of the programs regarding the specification.
- Verification of the integrated final system with respect to the global specification.
- Activities: Validation, Verification, Testing, Consistency- and Completeness-Check
- Tool support needed!

Requirements

- The global specification describes, as exact as possible, what must be done.
- **Abstraction of the *how***
  Advantages
  - apriori: Reference document, compact and legible.
  - aposteriori: Possibility to follow and document design decisions $\rightsquigarrow$ **traceability, reusability, maintenance**.
- Problem: Size and complexity of the systems.

Principles to be supported

- Refinement principle: Abstraction levels
- Structuring mechanisms: Decomposition and modularization techniques
- Object orientation
- Verification and validation concepts

Requirements Description ⇝ Specification Language

- Choice of the specification technique depends on the System. Frequently more than a single specification technique is needed. (What – How).
- Type of Systems:
  Pure function oriented (I/O), reactive- embedded- real time-systems.
- Problem : Universal Specification Technique (UST)
  difficult to understand, ambiguities, tools, size . . .
  e.g. UML
- Desired: Compact, legible and exact specifications

Here: functional specification techniques

Formal Specifications

- A specification in a formal specification language defines all the possible behaviors of the specified system.
- 3 Aspects: Syntax, Semantics, Inference System
  - Syntax: What's allowed to write: Text with structure, Properties often described by formulas from a logic, e.g equational logic.
  - Semantics: Which models are associated with the specification, $\rightsquigarrow$ Notion of models.
  - Inference System: Consequences (Derivation) of properties of the system. $\rightsquigarrow$ Notion of consequence.

Formal Specifications

- Two main classes:

Model oriented	-	-	Property oriented
(constructive)			(declarative)
e.g.VDM, Z, ASM			signature (functions, predicates)
Construction of a non-ambiguous model from available data structures and construction rules			Properties (formulas, axioms)
			models
			algebraic specification
Concept of correctness			AFFIRM, OBJ, ASF, HOL,. . .

- Operational specifications:
  Petri nets, process algebras, automata based (SDL).

Tool support

- Syntactic support (grammars, parser,...)
- Verification: theorem proving (proof obligations)
- Prototyping (executable specifications)
- Code generation (out of the specifications generate C code)
- Testing (from the specification generate test cases for the program)

Desired:

To generate the tools out of the syntax and semantics of the specification language

Example: declarative

Example 1.1. Restricted logic: e.g. equational logic

- Axioms: $\forall X\ t_1 = t_2 \qquad t_1, t_2$ terms.
- Rules: Equals are replaced with equals. (directed).
- Terms $\approx$ names for objects (identifier), structuring, construction of the object.
- Abstraction: Terms as elements of an algebra, term algebra.

Stack: algebraic specification

Example 1.2. Elements of an algebraic specification: Signature (sorts (types), operation names with arities), Axioms (often only equations)

SPEC STACK
USING NATURAL, BOOLEAN "Names of known SPECs"
SORT stack "Principal type"
OPS init : → stack "Constant of the type *stack*, empty stack"
push : stack nat → stack
pop : stack → stack
top : stack → nat
is_empty? : stack → bool
stack_error : → stack
nat_error : → nat

(Signature fixed)

Axioms for Stack

FORALL s : stack n : nat
AXIOMS

is_empty? (init) = true
is_empty? (push (s, n)) = false
pop (init) = stack_error
pop (push (s, n)) = s
top (init) = nat_error
top (push (s,n)) = n

Terms or expressions: top (push (push (init, 2), 3)) “means” 3

Semantics? Operationalization?

Apply equations as rules from left to right $\leadsto$

Notion of rules and rewriting

Example: Sorting of lists over arbitrary types

Example 1.3.

$$\text{Formal} :: \left\{ \begin{array}{ll} \text{spec} & \text{ELEMENT} \\ \text{use} & \text{BOOL} \\ \text{sorts} & \text{elem} \\ \text{ops} & . \leq . : \text{elem}, \text{elem} \to \text{bool} \\ \text{eqns} & x \leq x = \text{true} \\ & \text{imp}(x \leq y \text{ and } y \leq z, x \leq z) = \text{true} \\ & x \leq y \text{ or } y \leq x = \text{true} \end{array} \right.$$

Example (Cont.)

spec	LIST[ELEMENT]
use	ELEMENT
sorts	list
ops	nil $:\to$ list
	. : elem, list $\to$ list ("infix")
	insert : elem, list $\to$ list
	insertsort : list $\to$ list
	case : bool, list, list $\to$ list
	sorted : list $\to$ bool

Example (Cont.)

eqns $\text{case}(\text{true}, l_1, l_2) = l_1$
$\text{case}(\text{false}, l_1, l_2) = l_2$

$\text{insert}(x, \text{nil}) = x.\text{nil}$
$\text{insert}(x, y.l) = \text{case}(x \leq y, x.y.l, y.\,\text{insert}(x, l))$

$\text{insertsort}(\text{nil}) = \text{nil}$
$\text{insertsort}(x.l) = \text{insert}(x, \text{insertsort}(l))$

$\text{sorted}(\text{nil}) = \text{true}$
$\text{sorted}(x.\text{nil}) = \text{true}$
$\text{sorted}(x.y.l) = \text{if } x \leq y \text{ then sorted}(y.l) \text{ else false}$

Property: $\text{sorted}(\text{insertsort}(l)) = \text{true}$

Syntax

Aspects of syntax

- used to designate things and express facts
- terms and formulas are formed from variables and function symbols
- function symbols map a tupel of terms to another term
- constant symbols: no arguments
- constant can be seen as functions with zero arguments
- predicate symbols are considered as boolean functions
- set of variables

Syntax (cont.)

Example 1.4. Natural Numbers

- constant symbol: 0
- function symbol suc : $\mathbb{N} \to \mathbb{N}$
- function symbol plus : $\mathbb{N} \times \mathbb{N} \to \mathbb{N}$
- function symbol . . .

Syntax of propositional logic

Definition 1.5. *Symbols*

- $\mathcal{V} = \{a, b, c, \ldots\}$ *is a set of propositional variables*
- *two function symbols:* $\neg$ *and* $\rightarrow$

Definition 1.6. *Language*

- *each* $P \in \mathcal{V}$ *is a formula*
- *if* ϕ *is a formula, then* $\neg\phi$ *is a formula*
- *if* ϕ *and* ψ *are formulas, then* $\phi \rightarrow \psi$ *is a formula*

Semantics

Purpose

- syntax only specifies the structure of terms and formulas
- symbols and terms are assigned a meaning
- variables are assigned a value
- in particular, propositional variables are assigned a truth value

Bottom-Up Approach

- assignments give variables a value
- terms/formulas are evaluated based on the meaning of the function symbols

Interpretations/Structures

Definition 1.7. *Assignment in Propositional Logic*
A variable assignment in propositionan logic is a mapping

- $I : \mathcal{V} \rightarrow \{\text{true}, \text{false}\}$

Definition 1.8. *Valuation of Propositional Logic*
The valuation V takes an assignment I and a formula and yiels a true or false:

- *if* $\phi \in \mathcal{V}$*:* $V(\phi) = I(\phi)$
- $V(\neg\phi) = f_{\neg}(V(\phi))$
- $V(\phi \rightarrow \psi) = f_{\rightarrow}(V(\phi), V(\psi))$

where

$f_{\neg}$	
false	true
true	false

$f_{\rightarrow}$	false	true
false	true	true
true	false	true

Problem 1.9. Is V a well defined function?

Validity

Definition 1.10. *Validity of formulas in propositional logic*

- *a formula ϕ is valid if $VI\phi$ evaluates to* true *for all assigments I*
- *notation:* $\models \phi$

Example 1.11. Tautology in Propositional Logic

- $\phi = a \vee \neg a$ (where $a \in \mathcal{V}$) is valid
  - $I(a) =$ false: $V(a \vee \neg a) =$ true
  - $I(a) =$ true: $V(a \vee \neg a) =$ true

Syntactic Sugar

Purpose

- additions to the language that do not affect its expressiveness
- more practical way of description

Example 1.12. Abbreviations in Propositional Logic

- *True* denotes $\phi \rightarrow \phi$
- *False* denotes $\neg$*True*
- $\phi \vee \psi$ denotes $(\neg\phi) \rightarrow \psi$
- $\phi \wedge \psi$ denotes $\neg((\neg\phi) \vee (\neg\psi))$
- $\phi \leftrightarrow \psi$ denotes $((\phi \rightarrow \psi) \wedge (\psi \rightarrow \phi))$

Proof Systems/Logical Calculi: Introduction

General Concept

- purely syntactical manipulations based on designated transformation rules
- starting point: set of formulas, often a given set of axioms
- deriving new formulas by deduction rules from given formulas Γ
- ϕ is *provable* from Γ if ϕ can be obtained by a finite number of derivation steps assuming the formulas in Γ
- notation: $\Gamma \vdash \phi$ means ϕ is *provable* from Γ
- notation: $\vdash \phi$ means ϕ is *provable* from a given set of axioms

Proof System Styles

Hilbert Style

- easy to understand
- hard to use

Natural Deduction

- easy to use
- hard to understand

- ...

Hilbert-Style Deduction Rules

Definition 1.13. *Deduction Rule*

- *deduction rule d is a $n+1$-tuple*

$$\frac{\phi_1 \quad \cdots \quad \phi_n}{\psi}$$

- *formulas $\phi_1 \ldots \phi_n$, called premises of rule*
- *formula ψ, called conclusion of rule*

Hilbert-Style Proofs

Definition 1.14. *Proof*

- *let D be a set of deduction rules, including the axioms as rules without premisses*
- *proofs in D are (natural) trees such that*
  - *axioms are proofs*
  - *if $P_1, \ldots, P_n$ are proofs with roots $\phi_1 \ldots \phi_n$ and* $\dfrac{\phi_1 \cdots \phi_n}{\psi}$ *is in D, then* $\dfrac{P_1 \cdots P_n}{\psi}$ *is a proof in D*
- *can also be written in a line-oriented style*

Hilbert-Style Deduction Rules

Axioms

- let Γ be a set of axioms, $\psi \in \Gamma$, then $\frac{}{\psi}$ is a proof
- axioms allow to construct trivial proofs

Rule example

- Modus Ponens: $\frac{\phi \rightarrow \psi, \quad \phi}{\psi}$
- if $\phi \rightarrow \psi$ and ϕ have already been proven, ψ can be deduced

Proof Example

Example 1.15. Hilbert Proof

- language formed with the four proposition symbols P, Q, R, S
- axioms: P, Q, $Q \rightarrow R$, $P \rightarrow (R \rightarrow S)$

$$\dfrac{\dfrac{\dfrac{}{P \rightarrow (R \rightarrow S)} \quad \dfrac{}{P}}{R \rightarrow S} \qquad \dfrac{\dfrac{}{Q \rightarrow R} \quad \dfrac{}{Q}}{R}}{S}$$

Hilbert Calculus for Propositional Logic

Definition 1.16. *Axioms of Propositional Logic*
All instantiations of the following schemas:

- $A \rightarrow (B \rightarrow A)$
- $(A \rightarrow (B \rightarrow C)) \rightarrow ((A \rightarrow B) \rightarrow (A \rightarrow C))$
- $(\neg B \rightarrow \neg A) \rightarrow ((\neg B \rightarrow A) \rightarrow B)$
- *where A, B, C are arbitrary propositions*

Rules: All instantiations of modus ponens.

Natural Deduction

Motivation

- introducing a hypothesis is a natural step in a proof
- Hilbert proofs do not permit this directly
- can be only encoded by using $\longrightarrow$
- proofs are much longer and not very natural

Natural Deduction

- alternative definition where introduction of a hypothesis is a deduction rule
- deduction step can modify not only the proven propositions but also the assumptions Γ

Natural Deduction Rules

Definition 1.17. *Natural Deduction Rule*

- *deduction rule d is a $n+1$-tuple*

$$\frac{\Gamma_1 \vdash \phi_1 \quad \cdots \quad \Gamma_n \vdash \phi_n}{\Gamma \vdash \psi}$$

- *pairs of Γ (set of formulas) and ϕ (formulas): sequents*
- *proof: tree of sequents with rule instantiations as nodes*

Natural Deduction Rules

Natural Deduction Rules

- rich set of rules
- elimination rules eliminate a logical symbol from a premise
- introduction rules introduce a logical symbol into the conclusion
- reasoning from assumptions
- Assumption Introduction, Assumption weakening:

$$\frac{}{\Gamma \vdash \phi}\ \phi \in \Gamma \qquad\qquad \frac{\Gamma \vdash \phi}{\Gamma, \psi \vdash \phi}$$

Natural Deduction Rules

Definition 1.18. *Natural Deduction Rules for Propositional Logic*

- $\vee$-*introduction*

$$\frac{\Gamma \vdash \phi}{\Gamma \vdash \phi \vee \psi} \qquad \frac{\Gamma \vdash \psi}{\Gamma \vdash \phi \vee \psi}$$

- $\vee$-*elimination*

$$\frac{\Gamma \vdash \phi \vee \psi \qquad \Gamma, \phi \vdash \xi \qquad \Gamma, \psi \vdash \xi}{\Gamma \vdash \xi}$$

- $\rightarrow$-*introduction*

$$\frac{\Gamma, \phi \vdash \psi}{\Gamma \vdash \phi \rightarrow \psi}$$

- $\rightarrow$-*elimination*

$$\frac{\Gamma \vdash \phi \rightarrow \psi \qquad \Gamma \vdash \phi}{\Gamma \vdash \psi}$$

Natural Deduction Example

Example 1.19. $\{A \to C, B \to C\} \vdash (A \vee B) \to C$

$$\dfrac{\dfrac{\dfrac{}{\Gamma \vdash A \vee B} \qquad \dfrac{\dfrac{}{\Gamma, A \vdash A \to C} \qquad \dfrac{}{\Gamma, A \vdash A}}{\Gamma, A \vdash C} \qquad \dfrac{\cdots}{\Gamma, B \vdash C}}{\Gamma := \{A \to C, B \to C, A \vee B\} \vdash C}}{\{A \to C, B \to C\} \vdash (A \vee B) \to C}$$

Summary

Specification and verification

- Algebraic specification - Functional specification

Theorem-Proving Fundamentals

- syntax: symbols, terms, formulas
- semantics: (mathematical structures,) variable assigments, denotations for terms and formulas
- proof system/(logical) calculus: axioms, deduction rules, proofs, theories

Fundamental Principle of Logic: “Establish truth by calculation” (APH, 2010)

Chapter 2

Functional Programming:Isabelle

Overview of Isabelle/HOL

System Architecture

Isabelle generic theorem prover

System Architecture

Isabelle/HOL	Isabelle instance for HOL
Isabelle	generic theorem prover

System Architecture

ProofGeneral	(X)Emacs based interface
Isabelle/HOL	Isabelle instance for HOL
Isabelle	generic theorem prover

HOL

HOL = Higher-Order Logic

HOL

HOL = Higher-Order Logic

HOL = Functional programming + Logic

HOL

HOL = Higher-Order Logic

HOL = Functional programming + Logic

HOL has

- datatypes
- recursive functions
- logical operators ($\wedge$, $\longrightarrow$, $\forall$, $\exists$, . . .)

HOL

HOL = Higher-Order Logic

HOL = Functional programming + Logic

HOL has

- datatypes
- recursive functions
- logical operators ($\wedge$, $\longrightarrow$, $\forall$, $\exists$, ...)

HOL is a programming language!

HOL

HOL = Higher-Order Logic

HOL = Functional programming + Logic

HOL has

- datatypes
- recursive functions
- logical operators ($\wedge$, $\longrightarrow$, $\forall$, $\exists$, ...)

HOL is a programming language!

Higher-order = functions are values, too!

Formulae

Syntax (in decreasing priority):

$$\begin{array}{lllll} form & ::= & (form) & \mid\ term = term & \mid\ \neg form \\ & \mid & form \wedge form & \mid\ form \vee form & \mid\ form \longrightarrow form \\ & \mid & \forall x.\ form & \mid\ \exists x.\ form & \end{array}$$

Formulae

Syntax (in decreasing priority):

$$
\begin{array}{lcllll}
\mathit{form} & ::= & (\mathit{form}) & \mid\ \mathit{term} = \mathit{term} & \mid\ \neg \mathit{form} \\
 & \mid & \mathit{form} \wedge \mathit{form} & \mid\ \mathit{form} \vee \mathit{form} & \mid\ \mathit{form} \longrightarrow \mathit{form} \\
 & \mid & \forall x.\ \mathit{form} & \mid\ \exists x.\ \mathit{form} &
\end{array}
$$

Examples

- $\neg A \wedge B \vee C \ \equiv\ ((\neg A) \wedge B) \vee C$

Formulae

Syntax (in decreasing priority):

$$
\begin{array}{llllll}
form & ::= & (form) & \mid\ term = term & \mid\ \neg form \\
 & \mid & form \wedge form & \mid\ form \vee form & \mid\ form \longrightarrow form \\
 & \mid & \forall x.\ form & \mid\ \exists x.\ form &
\end{array}
$$

Examples

- $\neg A \wedge B \vee C \equiv ((\neg A) \wedge B) \vee C$
- $A = B \wedge C \equiv (A = B) \wedge C$

Formulae

Syntax (in decreasing priority):

$$
\begin{array}{lllll}
\mathit{form} & ::= & (\mathit{form}) & \mid\ \mathit{term} = \mathit{term} & \mid\ \neg\mathit{form} \\
& \mid & \mathit{form} \wedge \mathit{form} & \mid\ \mathit{form} \vee \mathit{form} & \mid\ \mathit{form} \longrightarrow \mathit{form} \\
& \mid & \forall x.\ \mathit{form} & \mid\ \exists x.\ \mathit{form} &
\end{array}
$$

Examples

- $\neg A \wedge B \vee C \equiv ((\neg A) \wedge B) \vee C$
- $A = B \wedge C \equiv (A = B) \wedge C$
- $\forall x.\ P\ x \wedge Q\ x \equiv \forall x.\ (P\ x \wedge Q\ x)$

Formulae

Syntax (in decreasing priority):

$$
\begin{array}{lllll}
\mathit{form} & ::= & (\mathit{form}) & \mid \ \mathit{term} = \mathit{term} & \mid \ \neg \mathit{form} \\
 & \mid & \mathit{form} \wedge \mathit{form} & \mid \ \mathit{form} \vee \mathit{form} & \mid \ \mathit{form} \longrightarrow \mathit{form} \\
 & \mid & \forall x.\ \mathit{form} & \mid \ \exists x.\ \mathit{form} &
\end{array}
$$

Examples

- $\neg A \wedge B \vee C \equiv ((\neg A) \wedge B) \vee C$
- $A = B \wedge C \equiv (A = B) \wedge C$
- $\forall x.\ P\ x \wedge Q\ x \equiv \forall x.\ (P\ x \wedge Q\ x)$

Scope of quantifiers: as far to the right as possible

Formulae

Abbreviation: $\forall x\, y.\; P\, x\, y \;\equiv\; \forall x.\, \forall y.\; P\, x\, y$

Formulae

Abbreviation: $\forall x\, y.\ P\ x\ y \;\equiv\; \forall x.\ \forall y.\ P\ x\ y \qquad (\forall, \exists, \lambda, \ldots)$

Formulae

Abbreviation: $\forall x\, y.\ P\ x\ y \;\equiv\; \forall x.\ \forall y.\ P\ x\ y \qquad (\forall, \exists, \lambda, \ldots)$

Parentheses:

- $\wedge$, $\vee$ and $\longrightarrow$ associate to the right:
  $A \wedge B \wedge C \;\equiv\; A \wedge (B \wedge C)$

Formulae

Abbreviation: $\forall x\ y.\ P\ x\ y \ \equiv\ \forall x.\ \forall y.\ P\ x\ y \qquad (\forall, \exists, \lambda, \ldots)$

Parentheses:

- $\wedge$, $\vee$ and $\longrightarrow$ associate to the right:
  $A \wedge B \wedge C \ \equiv\ A \wedge (B \wedge C)$
- $A \longrightarrow B \longrightarrow C \ \equiv\ A \longrightarrow (B \longrightarrow C) \not\equiv (A \longrightarrow B) \longrightarrow C$ **!**

Warning

Quantifiers have low priority and need to be parenthesized:

$$! \quad P \wedge \forall x.\ Q\ x \leadsto P \wedge (\forall x.\ Q\ x) \quad !$$

Types and Terms

Types

Syntax:

$$\begin{array}{lll}\tau & ::= & (\tau) \\ & | & \textit{bool} \mid \textit{nat} \mid \ldots \qquad \text{base types}\end{array}$$

Types

Syntax:

$$\begin{array}{lll} \tau & ::= & (\tau) \\ & | & \textit{bool} \mid \textit{nat} \mid \ldots \qquad \text{base types} \\ & | & \textit{'a} \mid \textit{'b} \mid \ldots \qquad \text{type variables} \end{array}$$

Types

Syntax:

$$
\begin{array}{lcll}
\tau & ::= & (\tau) & \\
 & | & \textit{bool} \mid \textit{nat} \mid \ldots & \text{base types} \\
 & | & \textit{'a} \mid \textit{'b} \mid \ldots & \text{type variables} \\
 & | & \tau \Rightarrow \tau & \text{total functions}
\end{array}
$$

Types

Syntax:

τ	$::=$	(τ)	
	$\mid$	*bool* $\mid$ *nat* $\mid \dots$	base types
	$\mid$	*'a* $\mid$ *'b* $\mid \dots$	type variables
	$\mid$	$\tau \Rightarrow \tau$	total functions
	$\mid$	$\tau \times \tau$	pairs (ascii: $*$)

Types

Syntax:

τ	::=	(τ)	
	$\mid$	*bool* $\mid$ *nat* $\mid \ldots$	base types
	$\mid$	*'a* $\mid$ *'b* $\mid \ldots$	type variables
	$\mid$	$\tau \Rightarrow \tau$	total functions
	$\mid$	$\tau \times \tau$	pairs (ascii: $*$)
	$\mid$	τ *list*	lists

Types

Syntax:

τ	::=	(τ)	
	$\mid$	*bool* $\mid$ *nat* $\mid \ldots$	base types
	$\mid$	*'a* $\mid$ *'b* $\mid \ldots$	type variables
	$\mid$	$\tau \Rightarrow \tau$	total functions
	$\mid$	$\tau \times \tau$	pairs (ascii: $*$)
	$\mid$	τ *list*	lists
	$\mid$	$\ldots$	user-defined types

Types

Syntax:

τ	$::=$	(τ)	
	$\mid$	*bool* $\mid$ *nat* $\mid \ldots$	base types
	$\mid$	*'a* $\mid$ *'b* $\mid \ldots$	type variables
	$\mid$	$\tau \Rightarrow \tau$	total functions
	$\mid$	$\tau \times \tau$	pairs (ascii: $*$)
	$\mid$	τ *list*	lists
	$\mid$	$\ldots$	user-defined types

Parentheses: $T1 \Rightarrow T2 \Rightarrow T3 \quad \equiv \quad T1 \Rightarrow (T2 \Rightarrow T3)$

Terms: Basic syntax

Syntax:

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

Terms: Basic syntax

Syntax:

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

Terms: Basic syntax

Syntax:

$term$	$::=$	$(term)$	
	$\mid$	a	constant or variable (identifier)
	$\mid$	$term\ term$	function application
	$\mid$	$\lambda x.\ term$	function “abstraction”
	$\mid$	$\ldots$	lots of syntactic sugar

Examples: *f (g x) y* *h (λx. f (g x))*

Terms: Basic syntax

Syntax:

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

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

Parantheses: $f\ a_1\ a_2\ a_3 \equiv ((f\ a_1)\ a_2)\ a_3$

λ-calculus on one slide

Informal notation: $t[x]$

λ-calculus on one slide

Informal notation: $t[x]$

- *Function application:*
  $f\ a$ is the call of function f with argument a

λ-calculus on one slide

Informal notation: $t[x]$

- *Function application:*
  $f\ a$ is the call of function f with argument a
- *Function abstraction:*
  $\lambda x.t[x]$ is the function with formal parameter x and body/result $t[x]$, i.e. $x \mapsto t[x]$.

λ-calculus on one slide

Informal notation: $t[x]$

- *Function application:*
  $f\ a$ is the call of function f with argument a
- *Function abstraction:*
  $\lambda x.t[x]$ is the function with formal parameter x and body/result $t[x]$, i.e. $x \mapsto t[x]$.
- *Computation:*
  Replace formal by actual parameter ("β-reduction"):
  $(\lambda x.t[x])\ a \quad \longrightarrow_\beta \quad t[a]$

λ-calculus on one slide

Informal notation: $t[x]$

- *Function application:*
  $f\ a$ is the call of function f with argument a
- *Function abstraction:*
  $\lambda x.t[x]$ is the function with formal parameter x and body/result $t[x]$, i.e. $x \mapsto t[x]$.
- *Computation:*
  Replace formal by actual parameter ("β-reduction"):
  $(\lambda x.t[x])\ a \quad \longrightarrow_\beta \quad t[a]$

Example: $(\lambda\ x.\ x + 5)\ 3 \quad \longrightarrow_\beta \quad (3 + 5)$

$\longrightarrow_\beta$ *in Isabelle: Don't worry, be happy*

Isabelle performs β-reduction automatically

Isabelle considers $(\lambda x.t[x])a$ and $t[a]$ equivalent

Terms and Types

Terms must be well-typed

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

Terms and Types

Terms must be well-typed

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

Notation: t `::` τ means t is a well-typed term of type τ.

Type inference

Isabelle automatically computes ("*infers*") the type of each variable in a term.

Type inference

Isabelle automatically computes ("*infers*") the type of each variable in a term.

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

Type inference

Isabelle automatically computes (“*infers*”) the type of each variable in a term.

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

User can help with type annotations inside the term.

Example: *f (x::nat)*

Currying

Thou shalt curry your functions

Currying

Thou shalt curry your functions

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

Currying

Thou shalt curry your functions

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

Advantage: *partial application* *f* a_1 with a_1 :: τ_1

Terms: Syntactic sugar

Some predefined syntactic sugar:

- *Infix:* +, -, *, #, @, . . .
- *Mixfix:* *if _ then _ else _*, *case _ of*, . . .

Terms: Syntactic sugar

Some predefined syntactic sugar:

- *Infix:* +, -, *, #, @, . . .
- *Mixfix:* *if _ then _ else _*, *case _ of*, . . .

Prefix binds more strongly than infix:

! $f\,x + y \equiv (f\,x) + y \not\equiv f\,(x + y)$ **!**

Terms: Syntactic sugar

Some predefined syntactic sugar:

- *Infix:* +, -, *, #, @, ...
- *Mixfix:* *if _ then _ else _*, *case _ of*, ...

Prefix binds more strongly than infix:

! $f\ x + y \equiv (f\ x) + y \neq f\ (x + y)$ **!**

Enclose *if* and *case* in parentheses:

! *(if _ then _ else _)* **!**

Base types: bool, nat, list

Type bool

Formulae = terms of type *bool*

Type bool

Formulae = terms of type *bool*

True :: *bool*
False :: *bool*
$\wedge, \vee, \ldots$ :: *bool* $\Rightarrow$ *bool* $\Rightarrow$ *bool*
$\vdots$

Type bool

Formulae = terms of type *bool*

True :: *bool*
False :: *bool*
$\wedge, \vee, \ldots$ :: *bool* $\Rightarrow$ *bool* $\Rightarrow$ *bool*
⋮

if-and-only-if: =

Type nat

$0 :: nat$
$Suc :: nat \Rightarrow nat$
$+, *, \ldots :: nat \Rightarrow nat \Rightarrow nat$
$\vdots$

Type nat

$0 :: nat$
$Suc :: nat \Rightarrow nat$
$+, *, \ldots :: nat \Rightarrow nat \Rightarrow nat$
$\vdots$

! Numbers and arithmetic operations are overloaded:
$0,1,2,\ldots :: 'a, \quad + :: \; 'a \Rightarrow 'a \Rightarrow 'a$

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

Type nat

0 :: nat
Suc :: nat ⇒ nat
*+, *, ... :: nat ⇒ nat ⇒ nat*
⋮

! Numbers and arithmetic operations are overloaded:
0,1,2,... :: 'a, + :: 'a ⇒ 'a ⇒ 'a

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

... unless the context is unambiguous: *Suc z*

Type list

- *[]*: empty list
- *x # xs*: list with first element *x* ("*head*")
  and rest *xs* ("*tail*")
- Syntactic sugar: *[x_1,...,x_n]*

Type list

- *[]*: empty list
- *x # xs*: list with first element *x* ("*head*")
  and rest *xs* ("*tail*")
- Syntactic sugar: $[x_1, \ldots, x_n]$

Large library:
hd, *tl*, *map*, *length*, *filter*, *set*, *nth*, *take*, *drop*, *distinct*, ...

Don't reinvent, reuse!
$\leadsto$ `HOL/List.thy`

Isabelle Theories

Theory = Module

Syntax:

```
theory MyTh
imports ImpTh_1 ... ImpTh_n
begin
(declarations, definitions, theorems, proofs, ...)*
end
```

- $MyTh$: name of theory. Must live in file $MyTh$`.thy`
- $ImpTh_i$: name of *imported* theories. Import transitive.

Theory = Module

Syntax:

```
theory MyTh
imports ImpTh_1 ... ImpTh_n
begin
(declarations, definitions, theorems, proofs, ...)*
end
```

- $MyTh$: name of theory. Must live in file $MyTh$`.thy`
- $ImpTh_i$: name of *imported* theories. Import transitive.

Usually:

```
theory MyTh
imports Main
⋮
```

Proof General

An Isabelle Interface

by David Aspinall

Proof General

Customized version of (x)emacs:

- all of emacs (info: `C-h i`)
- Isabelle aware (when editing `.thy` files)
- mathematical symbols (“x-symbols”)

X-Symbols

Input of funny symbols in Proof General

- via menu (“X-Symbol”)
- via ascii encoding (similar to LaTeX): \<and>, \<or>, ...
- via abbreviation: /\, \/, -->, ...

x-symbol	∀	∃	λ	¬	∧	∨	⟶	⇒
ascii (1)	\<forall>	\<exists>	\<lambda>	\<not>	/\	\/	-->	=>
ascii (2)	ALL	EX	%	~	&	|		

(1) is converted to x-symbol, (2) stays ascii.

Demo: terms and types

An introduction to recursion and induction

A recursive datatype: toy lists

datatype *'a list* = *Nil* | *Cons* *'a* *('a list)*

A recursive datatype: toy lists

datatype *'a list* = *Nil* | *Cons 'a ('a list)*

***Nil*:** empty list

***Cons x xs*:** head *x :: 'a*, tail *xs :: 'a list*

A recursive datatype: toy lists

datatype *'a list* = *Nil* | *Cons* *'a* *('a list)*

***Nil*:** empty list

***Cons x xs*:** head *x* :: *'a*, tail *xs* :: *'a list*

A toy list: *Cons False (Cons True Nil)*

A recursive datatype: toy lists

datatype *'a list = Nil | Cons 'a ('a list)*

***Nil*:** empty list

***Cons x xs*:** head *x :: 'a*, tail *xs :: 'a list*

A toy list: *Cons False (Cons True Nil)*

Predefined lists: *[False, True]*

Structural induction on lists

$P\ xs$ holds for all lists *xs* if

Structural induction on lists

P xs holds for all lists *xs* if

- *P Nil*

Structural induction on lists

P xs holds for all lists *xs* if

- *P Nil*
- and for arbitrary *x* and *xs*, *P xs* implies *P (Cons x xs)*

A recursive function: append

Definition by *primitive recursion*:

primrec *app* :: *'a list* ⇒ *'a list* ⇒ *'a list* **where**
app Nil ys = ? |
app (Cons x xs) ys = ??

A recursive function: append

Definition by *primitive recursion*:

primrec *app* :: *'a list* $\Rightarrow$ *'a list* $\Rightarrow$ *'a list* **where**
app Nil ys = ? |
app (Cons x xs) ys = ??

1 rule per constructor
Recursive calls must drop the constructor $\Longrightarrow$ Termination

Concrete syntax

In `.thy` files:

Types and formulas need to be inclosed in "

Concrete syntax

In `.thy` files:
Types and formulas need to be inclosed in "

Except for single identifiers, e.g. *'a*

Concrete syntax

In `.thy` files:
Types and formulas need to be inclosed in "

Except for single identifiers, e.g. *'a*

" normally not shown on slides

Demo: append and reverse

Proofs

General schema:

lemma *name*: "..."
apply (...)
apply (...)
⋮
done

If the lemma is suitable as a simplification rule:

lemma *name*[simp]: "..."

Proof methods

- Structural induction
  - Format: *(induct x)*
    x must be a free variable in the first subgoal.
    The type of *x* must be a datatype.
  - Effect: generates 1 new subgoal per constructor
- Simplification and a bit of logic
  - Format: *auto*
  - Effect: tries to solve as many subgoals as possible using simplification and basic logical reasoning.

Top down proofs

Command

sorry

“completes” any proof.

Top down proofs

Command

sorry

"completes" any proof.

Allows top down development:

Assume lemma first, prove it later.

Some useful tools

Disproving tools

Automatic counterexample search by random testing:

quickcheck

Disproving tools

Automatic counterexample search by random testing:

quickcheck

Counterexample search via SAT solver:

nitpick

Finding theorems

1. Click on Find button
2. Input search pattern (e.g. *"_ & True"*)

Demo: Disproving and Finding

Isabelle's meta-logic

Basic constructs

Implication $\Longrightarrow$ (==>)

For separating premises and conclusion of theorems

Basic constructs

Implication $\Longrightarrow$ (==>)
For separating premises and conclusion of theorems

Equality $\equiv$ (==)
For definitions

Basic constructs

Implication $\Longrightarrow$ (==>)
For separating premises and conclusion of theorems

Equality $\equiv$ (==)
For definitions

Universal quantifier $\bigwedge$ (!!)
For binding local variables

Basic constructs

Implication $\Longrightarrow$ (==>)
For separating premises and conclusion of theorems

Equality $\equiv$ (==)
For definitions

Universal quantifier $\bigwedge$ (!!)
For binding local variables

Do not use *inside* HOL formulae

Notation

$$[\![A_1; \ldots ; A_n]\!] \Longrightarrow B$$

abbreviates

$$A_1 \Longrightarrow \ldots \Longrightarrow A_n \Longrightarrow B$$

Notation

$$[\![A_1; \ldots ; A_n]\!] \Longrightarrow B$$

abbreviates

$$A_1 \Longrightarrow \ldots \Longrightarrow A_n \Longrightarrow B$$

$$; \quad \approx \quad \text{“and”}$$

The proof state

$$1.\ \bigwedge x_1 \ldots x_p.\ [\![A_1; \ldots ; A_n]\!] \Longrightarrow B$$

$x_1 \ldots x_p$	Local constants
$A_1 \ldots A_n$	Local assumptions
B	Actual (sub)goal

Type and function definition in Isabelle/HOL

Type definition in Isabelle/HOL

Introducing new types

Keywords:

- **typedecl**: pure declaration
- **types**: abbreviation
- **datatype**: recursive datatype

typedecl

typedecl $name$

Introduces new “opaque” type $name$ without definition

typedecl

typedecl $name$

Introduces new “opaque” type $name$ without definition

Example:

typedecl *addr* — An abstract type of addresses

types

types $name = \tau$

Introduces an *abbreviation* $name$ for type τ

types

types $name = \tau$

Introduces an *abbreviation* $name$ for type τ

Examples:

types
name = string
('a,'b)foo = 'a list $\times$ *'b list*

types

types $name = \tau$

Introduces an *abbreviation* $name$ for type τ

Examples:

types
name = string
('a,'b)foo = 'a list $\times$ *'b list*

Type abbreviations are expanded immediately after parsing
Not present in internal representation and Isabelle output

datatype

The example

datatype *'a list = Nil | Cons 'a ('a list)*

Properties:

- Types: $Nil \;::\; 'a\ list$
  $Cons \;::\; 'a \Rightarrow 'a\ list \Rightarrow 'a\ list$
- Distinctness: $Nil \neq Cons\ x\ xs$
- Injectivity: $(Cons\ x\ xs = Cons\ y\ ys) = (x = y \wedge xs = ys)$

The general case

$$\textbf{datatype}\ (\alpha_1, \ldots, \alpha_n)\tau \ = \ C_1\ \tau_{1,1} \ldots \tau_{1,n_1} \ \mid \ \ldots \ \mid \ C_k\ \tau_{k,1} \ldots \tau_{k,n_k}$$

- *Types:* $C_i :: \tau_{i,1} \Rightarrow \cdots \Rightarrow \tau_{i,n_i} \Rightarrow (\alpha_1, \ldots, \alpha_n)\tau$
- *Distinctness:* $C_i \ \ldots \neq C_j \ \ldots$ if $i \neq j$
- *Injectivity:*
  $(C_i\ x_1 \ldots x_{n_i} = C_i\ y_1 \ldots y_{n_i}) = (x_1 = y_1 \wedge \ldots \wedge x_{n_i} = y_{n_i})$

The general case

$$\textbf{datatype}\ (\alpha_1,\ldots,\alpha_n)\tau \;=\; C_1\ \tau_{1,1}\ldots\tau_{1,n_1} \mid \ldots \mid C_k\ \tau_{k,1}\ldots\tau_{k,n_k}$$

- *Types:* $C_i :: \tau_{i,1} \Rightarrow \cdots \Rightarrow \tau_{i,n_i} \Rightarrow (\alpha_1,\ldots,\alpha_n)\tau$
- *Distinctness:* $C_i\ \ldots \neq C_j\ \ldots$ if $i \neq j$
- *Injectivity:*
  $(C_i\ x_1 \ldots x_{n_i} = C_i\ y_1 \ldots y_{n_i}) = (x_1 = y_1 \wedge \ldots \wedge x_{n_i} = y_{n_i})$

Distinctness and Injectivity are applied automatically
Induction must be applied explicitly

Function definition in Isabelle/HOL

Why nontermination can be harmful

How about $f\ x = f\ x + 1$?

Why nontermination can be harmful

How about $f\ x = f\ x + 1$?

Subtract $f\ x$ on both sides.

$$\Longrightarrow 0 = 1$$

Why nontermination can be harmful

How about $f\ x = f\ x + 1$?

Subtract $f\ x$ on both sides.

$$\Longrightarrow 0 = 1$$

! All functions in HOL must be total **!**

Function definition schemas in Isabelle/HOL

- Non-recursive with **definition**
  No problem

Function definition schemas in Isabelle/HOL

- Non-recursive with **definition**
  No problem
- Primitive-recursive with **primrec**
  Terminating by construction

Function definition schemas in Isabelle/HOL

- Non-recursive with **definition**
  No problem
- Primitive-recursive with **primrec**
  Terminating by construction
- Well-founded recursion with **fun**
  Automatic termination proof

Function definition schemas in Isabelle/HOL

- Non-recursive with **definition**
  No problem
- Primitive-recursive with **primrec**
  Terminating by construction
- Well-founded recursion with **fun**
  Automatic termination proof
- Well-founded recursion with **function**
  User-supplied termination proof

definition

Definition (non-recursive) by example

definition $sq :: nat \Rightarrow nat$ **where** $sq\ n = n * n$

Definitions: pitfalls

definition $prime :: nat \Rightarrow bool$ **where**
$prime\ p = (1 < p \wedge (m\ dvd\ p \longrightarrow m = 1 \vee m = p))$

Definitions: pitfalls

definition $prime :: nat \Rightarrow bool$ **where**
$prime\ p = (1 < p \wedge (m\ dvd\ p \longrightarrow m = 1 \vee m = p))$

Not a definition: free *m* not on left-hand side

Definitions: pitfalls

definition *prime* :: *nat* ⇒ *bool* **where**
$prime\ p = (1 < p \wedge (m\ dvd\ p \longrightarrow m = 1 \vee m = p))$

Not a definition: free *m* not on left-hand side

! Every free variable on the rhs must occur on the lhs **!**

Definitions: pitfalls

definition $prime :: nat \Rightarrow bool$ **where**
$prime\ p = (1 < p \wedge (m\ dvd\ p \longrightarrow m = 1 \vee m = p))$

Not a definition: free *m* not on left-hand side

! Every free variable on the rhs must occur on the lhs **!**

$prime\ p = (1 < p \wedge (\forall m.\ m\ dvd\ p \longrightarrow m = 1 \vee m = p))$

Using definitions

Definitions are not used automatically

Using definitions

Definitions are not used automatically

Unfolding the definition of *sq*:

apply*(unfold sq_def)*

primrec

The example

primrec *app :: 'a list* $\Rightarrow$ *'a list* $\Rightarrow$ *'a list* **where**

app Nil ys = ys |

app (Cons x xs) ys = Cons x (app xs ys)

The general case

If τ is a datatype (with constructors $C_1, \ldots, C_k$) then
$f :: \cdots \Rightarrow \tau \Rightarrow \cdots \Rightarrow \tau'$ can be defined by *primitive recursion*:

$$f\ x_1 \ldots (C_1\ y_{1,1} \ldots y_{1,n_1}) \ldots x_p \quad = \quad r_1 \mid$$

$$\vdots$$

$$f\ x_1 \ldots (C_k\ y_{k,1} \ldots y_{k,n_k}) \ldots x_p \quad = \quad r_k$$

The general case

If τ is a datatype (with constructors $C_1, \dots, C_k$) then
$f :: \dots \Rightarrow \tau \Rightarrow \dots \Rightarrow \tau'$ can be defined by *primitive recursion*:

$$
\begin{array}{lcl}
f\ x_1 \dots (C_1\ y_{1,1} \dots y_{1,n_1}) \dots x_p & = & r_1 \mid \\
\vdots & & \\
f\ x_1 \dots (C_k\ y_{k,1} \dots y_{k,n_k}) \dots x_p & = & r_k
\end{array}
$$

The recursive calls in r_i must be *structurally smaller*,
i.e. of the form $f\ a_1 \dots y_{i,j} \dots a_p$

nat is a datatype

datatype *nat = 0 | Suc nat*

nat is a datatype

datatype *nat* = *0* | *Suc nat*

Functions on *nat* definable by primrec!

primrec $f :: nat \Rightarrow \ldots$
$f\ 0 = \ldots$
$f(Suc\ n) = \ldots f\ n \ldots$

More predefined types and functions

Type option

datatype *'a option = None* | *Some 'a*

Type option

datatype *'a option = None | Some 'a*

Important application:

$\ldots \Rightarrow$ *'a option*	$\approx$	partial function:
None	$\approx$	no result
Some a	$\approx$	result *a*

Type option

datatype *'a option = None | Some 'a*

Important application:

$$\ldots \Rightarrow \textit{'a option} \quad \approx \quad \text{partial function:}$$

$$\textit{None} \quad \approx \quad \text{no result}$$

$$\textit{Some a} \quad \approx \quad \text{result } a$$

Example:

primrec *lookup :: 'k* $\Rightarrow$ *('k* $\times$ *'v) list* $\Rightarrow$ *'v option* **where**

Type option

datatype *'a option = None | Some 'a*

Important application:

$$\ldots \Rightarrow \textit{'a option} \quad \approx \quad \text{partial function:}$$

$$\textit{None} \quad \approx \quad \text{no result}$$

$$\textit{Some a} \quad \approx \quad \text{result } a$$

Example:

primrec *lookup :: 'k* $\Rightarrow$ *('k* $\times$ *'v) list* $\Rightarrow$ *'v option* **where**

lookup k [] = None

Type option

datatype *'a option = None | Some 'a*

Important application:

$$\ldots \Rightarrow \textit{'a option} \quad \approx \quad \text{partial function:}$$

$$\textit{None} \quad \approx \quad \text{no result}$$

$$\textit{Some a} \quad \approx \quad \text{result } a$$

Example:

primrec *lookup* :: *'k* $\Rightarrow$ *('k* $\times$ *'v) list* $\Rightarrow$ *'v option* **where**

lookup k [] = None |

lookup k (x#xs) =

(if fst x = k then Some(snd x) else lookup k xs)

case

Datatype values can be taken apart with *case* expressions:

(case xs of [] ⇒ . . . | y#ys ⇒ ... y ... ys ...)

case

Datatype values can be taken apart with *case* expressions:

(case xs of [] $\Rightarrow \ldots$ | y#ys $\Rightarrow$ ... y ... ys ...)

Wildcards:

(case xs of [] $\Rightarrow$ [] | y#_ $\Rightarrow$ [y])

case

Datatype values can be taken apart with *case* expressions:

$$(case\ xs\ of\ [] \Rightarrow \ldots \mid y\#ys \Rightarrow ...\ y\ ...\ ys\ ...)$$

Wildcards:

$$(case\ xs\ of\ [] \Rightarrow [] \mid y\#\_ \Rightarrow [y])$$

Nested patterns:

$$(case\ xs\ of\ [0] \Rightarrow 0 \mid [Suc\ n] \Rightarrow n \mid \_ \Rightarrow 2)$$

case

Datatype values can be taken apart with *case* expressions:

(case xs of [] ⇒ . . . | y#ys ⇒ ... y ... ys ...)

Wildcards:

(case xs of [] ⇒ [] | y#_ ⇒ [y])

Nested patterns:

(case xs of [0] ⇒ 0 | [Suc n] ⇒ n | _ ⇒ 2)

Complicated patterns mean complicated proofs!

case

Datatype values can be taken apart with *case* expressions:

(case xs of [] ⇒ . . . | y#ys ⇒ ... y ... ys ...)

Wildcards:

(case xs of [] ⇒ [] | y#_ ⇒ [y])

Nested patterns:

(case xs of [0] ⇒ 0 | [Suc n] ⇒ n | _ ⇒ 2)

Complicated patterns mean complicated proofs!

Needs *( )* in context

Proof by case distinction

If $t :: \tau$ and τ is a datatype

apply(*case_tac* t)

Proof by case distinction

If $t :: \tau$ and τ is a datatype

apply(*case_tac* t)

creates k subgoals

$$t = C_i \ x_1 \ldots x_p \Longrightarrow \ldots$$

one for each constructor C_i of type τ.

Demo: trees

fun

From primitive recursion to arbitrary pattern matching

Example: Fibonacchi

fun *fib* :: *nat* ⇒ *nat* **where**

fib 0 = 0 |
fib (Suc 0) = 1 |
fib (Suc(Suc n)) = fib (n+1) + fib n

Example: Separation

fun *sep :: 'a ⇒ 'a list ⇒ 'a list* **where**

sep a [] = [] |
sep a [x] = [x] |
sep a (x#y#zs) = x # a # sep a (y#zs)

Example: Ackermann

fun $ack :: nat \Rightarrow nat \Rightarrow nat$ **where**

$$
\begin{aligned}
&ack\ 0\ n = Suc\ n\ \mid \\
&ack\ (Suc\ m)\ 0 = ack\ m\ (Suc\ 0)\ \mid \\
&ack\ (Suc\ m)\ (Suc\ n) = ack\ m\ (ack\ (Suc\ m)\ n)
\end{aligned}
$$

Key features of fun

- Arbitrary pattern matching

Key features of fun

- Arbitrary pattern matching
- Order of equations matters

Key features of fun

- Arbitrary pattern matching
- Order of equations matters
- Termination must be provable
  by lexicographic combination of size measures

Size

- *size(n::nat) = n*

Size

- *size(n::nat) = n*
- *size(xs) = length xs*

Size

- *size(n::nat) = n*
- *size(xs) = length xs*
- *size* counts number of (non-nullary) constructors

Lexicographic ordering

Either the first component decreases, or it stays unchanged and the second component decreases:

Lexicographic ordering

Either the first component decreases, or it stays unchanged and the second component decreases:

$$(5,3) > (4,7) > (4,6) > (4,0) > (3,42) > \cdots$$

Lexicographic ordering

Either the first component decreases, or it stays unchanged and the second component decreases:

$$(5,3) > (4,7) > (4,6) > (4,0) > (3,42) > \cdots$$

Similar for tuples:

$$(5,6,3) > (4,12,5) > (4,11,9) > (4,11,8) > \cdots$$

Lexicographic ordering

Either the first component decreases, or it stays unchanged and the second component decreases:

$$(5,3) > (4,7) > (4,6) > (4,0) > (3,42) > \cdots$$

Similar for tuples:

$$(5,6,3) > (4,12,5) > (4,11,9) > (4,11,8) > \cdots$$

Theorem If each component ordering terminates, then their *lexicographic product* terminates, too.

Ackermann terminates

ack 0 n = Suc n

ack (Suc m) 0 = ack m (Suc 0)

ack (Suc m) (Suc n) = ack m (ack (Suc m) n)

Ackermann terminates

ack 0 n = Suc n

ack (Suc m) 0 = ack m (Suc 0)

ack (Suc m) (Suc n) = ack m (ack (Suc m) n)

because the arguments of each recursive call are lexicographically smaller than the arguments on the lhs.

Ackermann terminates

ack 0 n = Suc n

ack (Suc m) 0 = ack m (Suc 0)

ack (Suc m) (Suc n) = ack m (ack (Suc m) n)

because the arguments of each recursive call are lexicographically smaller than the arguments on the lhs.

Note: order of arguments not important for Isabelle!

Computation Induction

If $f :: \tau \Rightarrow \tau'$ is defined by **fun**, a special induction schema is provided to prove $P(x)$ for all $x :: \tau$:

Computation Induction

If $f :: \tau \Rightarrow \tau'$ is defined by **fun**, a special induction schema is provided to prove $P(x)$ for all $x :: \tau$:

for each equation $f(e) = t$,
prove $P(e)$ assuming $P(r)$ for all recursive calls $f(r)$ in t.

Computation Induction

If $f :: \tau \Rightarrow \tau'$ is defined by **fun**, a special induction schema is provided to prove $P(x)$ for all $x :: \tau$:

> for each equation $f(e) = t$,
> prove $P(e)$ assuming $P(r)$ for all recursive calls $f(r)$ in t.

Induction follows course of (terminating!) computation

Computation Induction: Example

fun *div2* :: *nat* ⇒ *nat* **where**
div2 0 = 0 |
div2 (Suc 0) = 0 |
div2(Suc(Suc n)) = Suc(div2 n)

Computation Induction: Example

fun *div2 :: nat* ⇒ *nat* **where**
div2 0 = 0 |
div2 (Suc 0) = 0 |
div2(Suc(Suc n)) = Suc(div2 n)

⇝ induction rule `div2.induct`:

$$\frac{P(0) \quad P(Suc\ 0) \quad P(n) \Longrightarrow P(Suc(Suc\ n))}{P(m)}$$

Demo: fun

Chapter 3

HOL:Foundations

Introduction

- Stands for Higher Order Logic
- Denotes both a logic and a system
- Logic is an evolution of Alonzo Church's Simple Theory of Types (1940)
- System is an evolution of LCF (1979)
- Intent of this lecture: give an overview of HOL

Some Logical History

- Frege was a logicist (math is a subset of logic)
- Proposed a system on which (he thought) all mathematics could be derived (in principle)
- Bertrand Russell found paradox in Frege's system
- Proposed the Ramified Theory of Types
- Wrote *Principia Mathematica* with Whitehead
- An attempt at developing basic mathematics completely formally

"My intellect never recovered from the strain"

Russell's Paradox

Definition

A set s does not contain itself if $s \notin s$

Fact

Consider $X = \{s \mid s \notin s\}$. X is the set of all sets that do not contain themselves.

- *If $X \in X$ then X does not contain itself, i.e., $X \notin X$*
- *If $X \notin X$ then X contains itself, i.e., $X \in X$*

So $X \in X$ iff $X \notin X$. Contradiction.

- Gottlob, we have a problem!

Type Theory

- Problem: even allowing the expression of the notion of sets that do not contain themselves leads to contradiction
- One solution: ban such self-referential expressions (so-called *vicious circles*)
- Russell's proposal: invent a hierarchy of types
- Elements of lower types could not be applied to elements of higher types
- Blocks the paradox because $X \in X$ no longer a well-formed expression

Type Theories

- Russell's Ramified Theory of Types was very complex
- Simplified by Frank Ramsey in 1920s
- A. Church used typed λ-calculus to give a sleek presentation (Simple Theory of Types 1940)
- An earlier attempt by Church used untyped λ-calculus as a foundation for mathematics. It was inconsistent.
- HOL is a version of Church's 1940 logic.
- Many other variants as well, *e.g.*, Calculus of Constructions

History of HOL Implementations

- Late 1960's : Dana Scott's Domain Theory
- Logic of Computable Functions: a (first order) logic for Scott's theory
- Implemented in Edinburgh LCF (mid-1970s)
- Early 1980's : Mike Gordon swapped Scott's logic for Church's
- Kept much of LCF implementation

Contemporary Implementations of HOL

- HOL-Light (Harrison)
- HOL-4 (Gordon, Slind, Norrish, others)
- Isabelle/HOL (Paulson,Nipkow)
- ProofPower (Arthan)
- reFLect (Intel)

Related systems:

- PVS (extension of Church's logic with dependent types and subtypes)
- ACL2 (built on Common Lisp subset)
- MIZAR (Tarski-Grothendieck set theory)

Page of Logic Implementations

For a collection of logic implementations see

`http://www.cs.ru.nl/~freek/digimath/index.html`

Motivation

- Higher-order logic (HOL) is an expressive foundation for
  mathematics: analysis, algebra, . . .
  computer science: program correctness, hardware verification, . . .
- Reasoning in HOL is classical.
- Still important: modeling of problems (now in HOL).
- Still important: deriving relevant reasoning principles.

(rev. 12275)

Motivation (2)

- HOL offers safety through strength:
  - small kernel of constants and axioms;
  - Safety via conservative (definitional) extensions.
- Contrast with
  - weak logics (e.g., propositional logic): can't define much;
  - axiomatic extensions: can lead to inconsistency

Bertrand Russell once likened the advantages of postulation over definition to the advantages of theft over honest toil!

(rev. 12275)

Alternatives to Isabelle/HOL

- We will use and focus on Isabelle/HOL.
- Could forgo the use of a meta-logic and employ alternatives, e.g., HOL system or PVS. Or use constructive alternatives such as Coq or Nuprl.
- Choice depends on culture and application.

Which Foundation?

- Set theory is often seen as the basis for mathematics.
  - Zermelo-Fraenkel, Bernays-Gödel, . . .
  - 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 is not a set. A class cannot belong to another class (let alone a set)!
- HOL as an alternative (Church 1940, Henkin 1950).
  - **Rationale:** one usually works with typed entities.
  - 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. —Larry Paulson
- Another alternative: category theory (Eilenberg, Mac Lane)

(rev. 12275)

Meaning of “Higher Order”

1st-order: quantification over individuals (0th-order objects).

$$\forall x,\, y.\, R(x, y) \longrightarrow R(y, x)$$

2nd-order: quantification over predicates and 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.

⋮

“higher order” $\rightsquigarrow$ union of all finite orders

(rev. 12275)

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., $list(bool)$
  - Isabelle supports polymorphic type definitions, e.g., $list(\alpha)$

- Terms: ($\mathcal{V}$ set of variables and $\mathcal{C}$ set of 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.

(rev. 12275)

Basic HOL Syntax (2)

- Constants are always supplied with types and include:

$$True, False : bool$$
$$\_ = \_ : \tau \Rightarrow \tau \Rightarrow bool \qquad \text{(for all types } \tau\text{)}$$
$$\_ \longrightarrow \_ : bool \Rightarrow bool \Rightarrow bool$$
$$\iota\_ : (\tau \Rightarrow bool) \Rightarrow \tau \qquad \text{(for all types } \tau\text{)}$$

- 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.

(rev. 12275)

HOL Semantics

- Intuitively an extension of many-sorted semantics with functions
  - FOL: structure is domain and functions/relations

    $$\langle \mathcal{D}, (f_i)_{i \in F}, (r_i)_{i \in R} \rangle$$

  - Many-sorted FOL: domains are sort-indexed

    $$\langle (\mathcal{D}_i)_{i \in S}, (f_i)_{i \in F}, (r_i)_{i \in R} \rangle$$

  - 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]).

(rev. 12275)

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 a nonempty set

Sub: If $X \in \mathcal{U}$ and $Y \neq \emptyset \subseteq X$, then $Y \in \mathcal{U}$

Prod: If $X, Y \in \mathcal{U}$ then $X \times Y \in \mathcal{U}$.

$X \times Y$ 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

(rev. 12275)

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 a nonempty 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 $\{T, F\}$.

(rev. 12275)

Definition 2 (Frame):

A frame is a collection $(\mathcal{D}_\alpha)_{\alpha \in \tau}$ with $\mathcal{D}_\alpha \in \mathcal{U}$, for $\alpha \in \tau$ and

- $\mathcal{D}_{bool} = \{T, F\}$
- $\mathcal{D}_{ind} = X$ 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.

Definition 3 (Interpretation):

An interpretation $\langle (\mathcal{D}_\alpha)_{\alpha \in \tau}, \mathcal{J} \rangle$ consists of a frame $(\mathcal{D}_\alpha)_{\alpha \in \tau}$ 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 the identity on $\mathcal{D}_\alpha$
- $\mathcal{J}(\longrightarrow)$ denotes the implication function over $\mathcal{D}_{bool}$, i.e.,

$$b \to b' = \begin{cases} F & \text{if } b = T \text{ and } b' = F \\ T & \text{otherwise} \end{cases}$$

- $\mathcal{J}(\iota_{(\alpha \Rightarrow bool) \Rightarrow \alpha}) \in (\mathcal{D}_\alpha \Rightarrow \mathcal{D}_{bool}) \Rightarrow \mathcal{D}_\alpha$ denotes the function

$$the(f) = \begin{cases} a & \text{if } f = (\lambda x. x = a) \\ y & \text{otherwise } (y \in \mathcal{D}_\alpha \text{ is arbitrary}) \end{cases}$$

(rev. 12275)

Definition 4 (Generalized Models):

An interpretation $\mathfrak{M} = \langle (\mathcal{D}_\alpha)_{\alpha \in \tau}, \mathcal{J} \rangle$ is a (general) model for HOL iff there is a binary function $\mathcal{V}^{\mathfrak{M}}$ such that

- for all type-indexed families of substitutions $\sigma = (\sigma_\alpha)_{\alpha \in \tau}$ and terms t of type α, $\mathcal{V}^{\mathfrak{M}}(\sigma, t) \in \mathcal{D}_\alpha$, and
- for all type-indexed families of substitutions $\sigma = (\sigma_\alpha)_{\alpha \in \tau}$,
  - (a) $\mathcal{V}^{\mathfrak{M}}(\sigma, x_\alpha) = \sigma_\alpha(x_\alpha)$
  - (b) $\mathcal{V}^{\mathfrak{M}}(\sigma, c) = \mathcal{J}(c)$, for c a (primitive) constant
  - (c) $\mathcal{V}^{\mathfrak{M}}(\sigma, s_{\alpha \Rightarrow \beta} t_\alpha) = \mathcal{V}^{\mathfrak{M}}(\sigma, s)\mathcal{V}^{\mathfrak{M}}(\sigma, t)$
    i.e., the value of the function $\mathcal{V}^{\mathfrak{M}}(\sigma, s)$ at the argument $\mathcal{V}^{\mathfrak{M}}(\sigma, t)$
  - (d) $\mathcal{V}^{\mathfrak{M}}(\lambda x_\alpha. t_\beta) =$ "the function from $\mathcal{D}_\alpha$ into $\mathcal{D}_\beta$ whose value for each $z \in \mathcal{D}_\alpha$ is $\mathcal{V}^{\mathfrak{M}}(\sigma[x \leftarrow z], t)$"

(rev. 12275)

Generalized Models - Facts (1)

- **If** $\mathfrak{M}$ is a general model and σ a substitution, **then** $\mathcal{V}^{\mathfrak{M}}(\sigma, t)$ is uniquely determined, for every term t. $\mathcal{V}^{\mathfrak{M}}(\sigma, t)$ is value of t in $\mathfrak{M}$ w.r.t. σ.
- Gives rise to the standard notion of satisfiability/validity:
  - We write $\mathcal{V}^{\mathfrak{M}}, \sigma \models \phi$ for $\mathcal{V}^{\mathfrak{M}}(\sigma, \phi) = T$.
  - ϕ is satisfiable in $\mathfrak{M}$ if $\mathcal{V}^{\mathfrak{M}}, \sigma \models \phi$, for some substitution σ.
  - ϕ is valid in $\mathfrak{M}$ if $\mathcal{V}^{\mathfrak{M}}, \sigma \models \phi$, for every substitution σ.
  - ϕ is valid (in the general sense) if ϕ is valid in every general model $\mathfrak{M}$.

(rev. 12275)

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 belong to $\mathcal{D}_{\alpha\Rightarrow\alpha}$ so that $\mathcal{V}^{\mathfrak{M}}(\sigma, \lambda x_\alpha.\, x)$ is defined.

closure under application:

- if $\mathcal{D}_N$ is set of 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 $k = \mathcal{V}^{\mathfrak{M}}(\sigma, \lambda x.\, f\,(f\,x\,x)\,y)$ where $\sigma(f) = p$ and $\sigma(y) = 5$.

(rev. 12275)

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.
- Analogous definitions for satisfiability and validity w.r.t. standard models.

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.
- Analogous definitions for satisfiability and validity w.r.t. standard models.

- We can now re-introduce HOL in Isabelle's meta-logic.

(rev. 12275)

Isabelle/HOL

The syntax of the core-language is introduced by:

consts

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)

(rev. 12275)

The Axioms of HOL (1)

axioms

refl : $"t = t"$

subst: $"[\![s = t;\ P(s)]\!] \Longrightarrow P(t)"$

ext: $"(\bigwedge x.\ f\ x = g\ x) \Longrightarrow (\lambda x.\ f\ x) = (\lambda x.\ g\ x)"$

impl: $"(P \Longrightarrow Q) \Longrightarrow P \longrightarrow Q"$

mp: $"[\![P \longrightarrow Q;\ P]\!] \Longrightarrow Q"$

iff : $"(P \longrightarrow Q) \longrightarrow (Q \longrightarrow P) \longrightarrow (P = Q)"$

True_or_False : $"(P = True) \vee (P = False)"$

the_eq_trivial : $"(THE\ x.\ x = a) = (a{::}'a)"$

(rev. 12275)

The Axioms of HOL (2)

Additionally, there is:

- universal α, β, and η congruence on terms (implicitly),
- the axiom of infinity, and
- the axiom of choice (Hilbert operator).
- This is the entire basis!

Core Definitions of HOL

defs

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	$\equiv THE\ z::'a.\ (P{=}True \longrightarrow z{=}x) \wedge (P{=}False \longrightarrow z{=}y)$

(rev. 12275)

Meta-theoretic Properties of HOL

Theorem 1 (Soundness of HOL, [And86]):

HOL is sound w.r.t. to general models.

$$\vdash_{HOL} \phi \quad \text{implies} \quad \phi \text{ is valid}$$

Theorem 2 (Completeness of HOL, [And86]):

- HOL is complete w.r.t. to general models.

$$\phi \text{ is valid} \quad \text{implies} \quad \vdash_{HOL} \phi$$

- HOL is complete w.r.t. to standard models.

Theorem 3 (HOL with infinity, [And86]):

- HOL+infinity is complete w.r.t. general models.
- HOL+infinity is incomplete w.r.t. standard models.

(rev. 12275)

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!)

(rev. 12275)

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.

References

[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.

[BN98] Franz Baader and Tobias Nipkow. *Term Rewriting and* All That. Cambridge University Press, 1998.

[Chu40] Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[Gen35] Gerhard Gentzen. Untersuchungen über das logische Schliessen. *Mathematische Zeitschrift*, 39:176–210, 405–431, 1935. English translation in [Sza69].

[GLT89] Jean-Yves Girard, Yves Lafont, and Paul Taylor. *Proofs and Types*. Cambridge University Press, 1989.

[GM93] Michael J. C. Gordon and Tom F. Melham, editors. *Introduction to HOL*. Cambridge University Press, 1993.

[HHPW96] Cordelia V. Hall, Kevin Hammond, Simon L. Peyton Jones, and Philipp Wadler. Type classes in Haskell. *ACM Transactions on Programming Languages and Systems*, 18(2):109–138, 1996.

[Höl90] Steffen Hölldobler. Conditional equational theories and complete sets of transformations. *Theoretical Computer Science*, 75(1&2):85–110, 1990.

[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.

[Mil78] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, 1978.

[Nip93] Tobias Nipkow. *Logical Environments*, chapter Order-Sorted Polymorphism in Isabelle, pages 164–188. Cambridge University Press, 1993.

[NN99] Wolfgang Naraschewski and Tobias Nipkow. Type inference verified: Algorithm $\mathcal{W}$ in isabelle/hol. *Journal of Automated Reasoning*, 23(3-4):299–318, 1999.

[Pau96] Lawrence C. Paulson. *ML for the Working Programmer*. Cambridge University Press, 1996.

[Pau03] Lawrence C. Paulson. *The Isabelle Reference Manual*. Computer Laboratory, University of Cambridge, March 2003.

[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.

[Sza69] M. E. Szabo. *The Collected Papers of Gerhard Gentzen*. North-Holland, 1969.

[Tho95] Simon Thompson. *Miranda: The Craft of Functional Programming*. Addison-Wesley, 1995.

[Tho99] Simon Thompson. *Haskell: The Craft of Functional Programming*. Addison-Wesley, 1999. Second Edition.

[vD80] Dirk van Dalen. *Logic and Structure*. Springer-Verlag, 1980. An introductory textbook on logic.

[Vel94] Daniel J. Velleman. *How to Prove It*. Cambridge University Press, 1994.

[vH67] Jean van Heijenoort, editor. *From Frege to Gödel: A Source Book in Mathematical Logic, 1879-193*. Harvard University Press, 1967. Contains translations of original works by David Hilbert.

[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.

[WR25] Alfred N. Whitehead and Bertrand Russell. *Principia Mathematica*, volume 1. Cambridge University Press, 1925. 2nd edition.

Higher-order Logic: Conservative Extensions

Outline

In the previous lecture, we have derived all well-known inference rules. There is now the need to scale up. Today we look at conservative theory extensions, an important method for this purpose.

In the weeks to come, we will look at how mathematics is encoded in the Isabelle/HOL library.

Conservative Theory Extensions: Basics

Terminology and 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 a set of axioms.

Definition 7 (consistent):

A theory T is consistent iff $False$ is not provable in T.

Definition 8 (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 $A \subseteq A'$.

(rev. 32934)

Definitions (Cont.)

Definition 9 (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 $|_\Sigma$ filters away all formulas not belonging to Σ.

Counterexample:

$$\overline{\forall f :: \alpha \Rightarrow \alpha.\, Y\ f = f\ (Y\ f)}\ \text{fix}$$

(rev. 32934)

Consistency Preserved

Lemma 1 (consistency):

If T' is a conservative extension of a consistent theory T, then

$$False \notin Th(T').$$

(rev. 32934)

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].

(rev. 32934)

Constant Definition

Definition 10 (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.

(rev. 32934)

Constant Definitions are Conservative

Lemma 2 (constant definitions):

A constant definition is a conservative extension.

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. π is a proof in T, i.e., $\phi \in Th(T)$.

(rev. 32934)

Side Conditions

Where are those side conditions needed? What goes wrong?

Simple example: Let $E \equiv \exists x :: \alpha.\ \exists y :: \alpha.\ x \neq y$ 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:

$$\begin{array}{ll} & c = c \qquad \text{holds by } \textit{refl} \\ \Longrightarrow & (\exists x :: \sigma.\ \exists y :: \sigma.\ x \neq y) = (\exists x :: \tau.\ \exists y :: \tau.\ x \neq y) \\ \Longrightarrow & \mathit{False} = \mathit{True} \\ \Longrightarrow & \mathit{False} \end{array}$$

Reconsider the definition of True.

(rev. 32934)

Constant Definition: Examples

Definitions of *True*, *False*, ¬, ∧, ∨, ∀, and ∃ 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:	¬ P	$\equiv P \longrightarrow False$
and_def:	P ∧ Q	$\equiv \forall R.\ (P \longrightarrow Q \longrightarrow R) \longrightarrow R$
or_def :	P ∨ Q	$\equiv \forall R.\ (P \longrightarrow R) \longrightarrow (Q \longrightarrow R) \longrightarrow R$

Recall that All (P) is equivalent to $\forall\ x.\ P\ x$ and
Ex(P) is equivalent to $\exists\ x.\ P\ x$.

(rev. 32934)

More Constant Definitions in Isabelle

let −**in**−, if −then−else, unique existence:

consts

```
Let  :: ['a, 'a ⇒ 'b] ⇒ 'b
If   :: [bool, 'a, 'a] ⇒ 'a
Ex1  :: ('a ⇒ bool) ⇒ bool
```

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)"
```

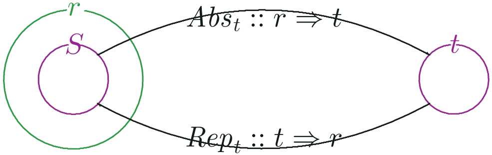
Note: ⇒ is function type arrow; recall syntax for [...] ⇒ ...

(rev. 32934)

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.



(rev. 32934)

Type Definition: Definition

Definition 11 (type definition):

Assume a theory $T = (\chi, \Sigma, A)$ and a type r and a term S of type $r \Rightarrow bool$.

A theory extension $T' = (\chi', \Sigma', A')$ of T is a type definition for type t (where t fresh), iff

$$\begin{array}{lcll} \chi' & = & \chi & \uplus \ \{t\}, \\ \Sigma' & = & \Sigma & \cup \ \{Abs_t :: r \Rightarrow t, Rep_t :: t \Rightarrow r\} \\ A' & = & A & \cup \ \{\forall x. Abs_t(Rep_t\, x) = x, \\ & & & \quad \forall x. S\, x \longrightarrow Rep_t(Abs_t\, x) = x\} \end{array}$$

Proof obligation $T \vdash \exists x.\, S\, x$ (inside HOL)

(rev. 32934)

Type Definitions are Conservative

Lemma 3 (type definitions):

A type definition is a conservative extension.

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?

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.

(rev. 32934)

Example: Typed Sets

General scheme, substituting $r \equiv \alpha \Rightarrow bool$ (α is any type variable), $t \equiv \alpha\ set$ (or set), $S \equiv \lambda x :: \alpha \Rightarrow bool.\ True$

$$
\begin{array}{lcll}
\chi' & = & \chi & \uplus\ \{set\}, \\
\Sigma' & = & \Sigma & \cup\ \{Abs_{set} :: (\alpha \Rightarrow bool) \Rightarrow \alpha\ set, \\
 & & & \quad Rep_{set} :: \alpha\ set \Rightarrow (\alpha \Rightarrow bool)\} \\
A' & = & A & \cup\ \{\forall x. Abs_{set}(Rep_{set}\ x) = x, \\
 & & & \quad \forall x. \qquad Rep_{set}(Abs_{set}\ x) = x\}
\end{array}
$$

Simplification since $S \equiv \lambda x.\ True$. Proof obligation: $(\exists x.\ S\ x)$ trivial since $(\exists x.\ True) = True$. Inhabitation is crucial!

(rev. 32934)

Sets: Remarks

Any function $f :: \tau \Rightarrow bool$ can be interpreted as a set of τ; f is called characteristic function. That's what $Abs_{set}\ f$ does; Abs_{set} is a wrapper saying "interpret f as set".

$S \equiv \lambda x. True$ and so S is trivial in this case.

(rev. 32934)

More Constants for Sets

For convenient use of sets, we define more constants:

$$\begin{aligned}\{x \mid f\,x\} &\cong Collect\ f = Abs_{set}\ f\\ x \in A &= (Rep_{set}\ A)\ x\\ A \cup B &= \{x \mid x \in A \vee x \in B\}\\ &\vdots\end{aligned}$$

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.

(rev. 32934)

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, $\lambda x :: \alpha.\ \lambda y :: \beta.\ x = a \wedge y = b$).
- Some functions of type $\alpha \Rightarrow \beta \Rightarrow bool$ represent pairs and others don't (e.g., the function $\lambda x.\ \lambda y.\ True$ does not represent a pair). The ones that do are are equal to $\lambda x :: \alpha.\ \lambda y :: \beta.\ x = a \wedge y = b$, for some a and b.

Type Definition for Pairs

This gives rise to a type definition where S is non-trivial:

$$
\begin{array}{rcl}
r & \equiv & \alpha \Rightarrow \beta \Rightarrow bool \\
S & \equiv & \lambda f :: \alpha \Rightarrow \beta \Rightarrow bool. \\
 & & \quad \exists a.\ \exists b.\ f = \lambda x :: \alpha.\ \lambda y :: \beta.\ x = a \wedge y = b \\
t & \equiv & \alpha \times \beta \qquad\qquad (\times \text{ infix})
\end{array}
$$

It is convenient to define a constant Pair_Rep (not to be confused with $Rep_\times$) as follows:
Pair_Rep a b = λx ::' a. λ y ::' b. x=a ∧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.

(rev. 32934)

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.

(rev. 32934)

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) "+" (infixr 10)
   = "{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?

(rev. 32934)

Summary

- We have presented a method to safely build up larger theories:
  - Constant definitions;
  - Type definitions.
- Subtle side conditions.
- A new type must be isomorphic to a "subset" of an existing type.

(rev. 32934)

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.

(rev. 32934)

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.

(rev. 32934)

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.

(rev. 32934)

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.

(rev. 32934)

Domains of Σ, Γ

The domain of Σ, denoted $dom(\Sigma)$, is $\{c \mid (c :: A) \in \Sigma \text{ for some } A\}$.

Likewise, the domain of Γ, denoted $dom(\Gamma)$, is
$\{x \mid (x :: A) \in \Gamma \text{ for some } A\}$.

Note the slight abuse of notation.

(rev. 32934)

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 $id \equiv \lambda x.x$.

(rev. 32934)

S

Here, S is any "predicate", i.e., a term of type $r \Rightarrow bool$, not necessarily a constant.

(rev. 32934)

Fresh t

The type constructor t must not occur in χ.

(rev. 32934)

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.

(rev. 32934)

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 u of type t for which $S\,u$ holds.

Isomorphism

The formulas

$$\forall x. Abs_t(Rep_t\, x) = x$$
$$\forall x. S\, x \longrightarrow Rep_t(Abs_t\, x) = x$$

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.

(rev. 32934)

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 f of type r defines a set whose elements are of type α; $Abs_{set}\ f$ is that set.

(rev. 32934)

$Collect$

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 ∈-Sign

We define

$$x \in A \quad = \quad (Rep_{set}\ A)\ x$$

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 ∈) 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)"

(rev. 32934)

Collect_mem_eq [simp]: "{x. x:A} = A"

These axioms can be translated into definitions as follows:

$$
\begin{aligned}
&a \in \{x \mid P\,x\} = P\,a \leadsto \\
&a \in (Collect\,P) = P\,a \leadsto \\
&a \in (Abs_{set}\,P) = P\,a \leadsto \\
&Rep_{set}(Abs_{set}\,P)\,a = P\,a \leadsto Rep_{set}(Abs_{set}\,P) = P
\end{aligned}
$$

The last step uses extensionality.

Now the second one:

$$
\begin{aligned}
&\{x \mid x \in A\} = A \leadsto \\
&\{x \mid (Rep_{set} A)\,x\} = A \leadsto \\
&Collect(Rep_{set} A) = A
\end{aligned}
$$

Ignoring some universal quantifications (these are implicit in Isabelle),

(rev. 32934)

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.

(rev. 32934)

$Rep_\times$

$Rep_\times$ would be the generic name for one of the two isomorphism-defining functions.

Since $Rep_\times$ cannot 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 $b :: \beta$.

(rev. 32934)

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

More Detailed Explanations 742

PLUS::[even,even] → even (**infixl** 56)
PLUS_def "op PLUS ≡λxy. 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.

(rev. 32934)

Recursive Type definitions

Types One, Numbers, Lists, Trees

- Using Constant Definition and Type Definition
- one: use subset of bool
- num: use subset of ind +Axiom of infinity
- lists: use subset of $(num \to \alpha) \times num$
- trees: use num
- recursive type definitions: use one, $\times$, +, α-tree
- Details in Melham (89): Automating Recursive Type Definitions in HOL

Chapter 4

Proof system of Isabelle/HOL

Methods and Rules

Formulas, sequents, and rules revisited

Propositions can represent:

- formulas, generalized sequents: lemmas/theorems to be proven
- rules: to be applied in a proof step
- proof (sub-)goals, i.e., open leaves in a proof tree

Example: from Lecture.thy

- SPEC, SCHEMATIC (Warning)
- ARULE
- GOAL

A proven lemma/theorem is automatically transformed into a rule. That is, the set of rules is not fixed in Isabelle/HOL.E.g. ARULE.

Variables

Six kinds of variables:

- (logical) variables bound by the logic-quantifiers
- (logical) variables bound by the meta-quantifier
- free (logical) variables
- schematic variables (in rules and proofs)
- type variables
- schematic type variables

Format of Goals and Rules

Format of Goals

- $\bigwedge x1...xk.\ [|A1; ...; Am|] \Longrightarrow C$
- xi are variables local to the subgoal (possibly none)
- Ai are called the assumptions (possibly none)
- C is called the conclusion
- usually first three types of variables sometimes also schematic variables.

Format of Rules

- $[|P1; ...; Pn|] \Longrightarrow Q$
- Pi are called the premises (possibly none)
- P1 is called the major premise
- Q is called the consequent (not standard)
- Schematic variables in Pi, Q.

Application of rules

Methods are commands to work on the proof state

In particular, methods allow to apply rules. Whereas the set of rules is not fixed, the basic methods are fixed in Isabelle/HOL.

Rule application:

- Applying rules is based on unification.
- Unification is done w.r.t. the schematic variables.
- The unifier is applied to the complete proof state!
- Unification may involve renaming of bound variables.

Example: (general idea of rule application)

- rule: $[|P1; P2|] \Longrightarrow Q$
- subgoal: $A \Longrightarrow C$
- if U unifies C and Q, then sufficient subgoals are:
- $U(A) \Longrightarrow U(P1), U(A) \Longrightarrow U(P2)$

Methods

Command: apply(method <parameters>)

Application of a rule to a subgoal depends on the method:

Methods are (for convenience) be classified into:

- introduction methods: decompose formulae to the right of $\Longrightarrow$
- elimination methods: decompose formulae to the left of $\Longrightarrow$

Method rule <rulename> :

- unify Q with C; fails if no unifier exists; otherwise unifier U
- remaining subgoals: For $i = 1, ..., n$
- $\bigwedge x1...xk.\ U([|A1;...;Am|] \Longrightarrow Pi)$
- Example GOAL

Methods

Method assumption:

- unify C with first possible Aj; fails if no Aj exists for unification
- subgoal is closed (discharged)
- Example GOAL

Method erule <rulename> :

- unify Q with C and simultanneously unify P1 with some Aj; fails if no unifier exists; otherwise unifier U
- remaining subgoals: For $i = 2, ..., n$
- $\bigwedge x1...xk.\ U([|A1; ...; Am \backslash Aj|] \Longrightarrow Pi)$
- Example GOAL

Methods

Method drule <rulename> :

- unify P1 some Aj; fails if no unifier exists; otherwise unifier U
- remaining subgoals:
- For $i = 2, ..., n$ $\quad \bigwedge x1...xk.\ U([|A1; ...; Am \backslash Aj|] \Longrightarrow Pi)$
- $\bigwedge x1...xk.\ U([|A1; ...; Am \backslash Aj; Q|] \Longrightarrow C)$
- Example C1

Method frule <rulename> :

- unify P1 some Aj; fails if no unifier exists; otherwise unifier U
- remaining subgoals:
- For $i = 2, ..., n$ $\quad \bigwedge x1...xk.\ U([|A1; ...; Am|] \Longrightarrow Pi)$
- $\bigwedge x1...xk.\ U([|A1; ...; Am; Q|] \Longrightarrow C)$
- Example C1

Methods

Method [edf]rule_tac x= term in <rule> :

- are similar to the version above but allow to influence the unification
- Example 5.8.2, p. 79, TAC
- FIXAX2

Method unfold <name_def> :

- unfolds the definition of a constant in all subgoals
- Example SPEC

Method induct_tac <freevar...> :

- uses the inductive definition of a function
- generates the corresponding subgoals

Fundamental rules of Isabelle/HOl

See IsabelleHOLMain, Sect. 2.2

Remark

- Safe rules preserve provability
- e.g. conjI, impI, notI, iffI, refl, ccontr, classical, conjE, disjE
- Unsafe rules can turn a provable goal into an unprovable one
- e.g. disjI1, disjI2, impE, iffD1, iffD2, notE
- $\leadsto$ Apply safe rules before unsafe ones

Example

- lemma UNSAFE: "$A \vee \neg A$"
- apply (rule disI1)
- sorry

An overview of theory Main

The structure of theory Main: p. 23

Set construction in Isabelle/HOL: Sect. 6

Natural numbers in Isabelle/HOL: Sect. 15

Remark

Working with theory Main:

- The programmer cannot know the complete library
- The “verificator” cannot know all rules.

Rewriting and simplification

taken from IsabelleTutorial, Sect. 3.1) »> slidesNipkow:

apply(simp add: eq1 . . . eqn)

»> Demo: MyDemo, Simp

Overview

- Term rewriting foundations
- Term rewriting in Isabelle/HOL
  - Basic simplification
  - Extensions

Term rewriting foundations

Term rewriting means . . .

Using equations $l = r$ from left to right

Term rewriting means . . .

Using equations $l = r$ from left to right

As long as possible

Term rewriting means . . .

Using equations $l = r$ from left to right

As long as possible

Terminology: equation $\rightsquigarrow$ *rewrite rule*

An example

Equations:

$$\begin{aligned} 0+n &= n & (1)\\ (Suc\ m)+n &= Suc\ (m+n) & (2)\\ (Suc\ m \le Suc\ n) &= (m \le n) & (3)\\ (0 \le m) &= True & (4) \end{aligned}$$

An example

Equations:

$$\begin{aligned} 0 + n &= n & (1)\\ (Suc\ m) + n &= Suc\ (m + n) & (2)\\ (Suc\ m \leq Suc\ n) &= (m \leq n) & (3)\\ (0 \leq m) &= True & (4) \end{aligned}$$

$$0 + Suc\ 0 \ \leq \ Suc\ 0 + x$$

Rewriting:

An example

Equations:

$$
\begin{array}{rcll}
0 + n & = & n & (1)\\
(Suc\ m) + n & = & Suc\ (m + n) & (2)\\
(Suc\ m \leq Suc\ n) & = & (m \leq n) & (3)\\
(0 \leq m) & = & True & (4)
\end{array}
$$

Rewriting:

$$
\begin{array}{rcll}
0 + Suc\ 0 & \leq & Suc\ 0 + x & \overset{(1)}{=}\\
Suc\ 0 & \leq & Suc\ 0 + x &
\end{array}
$$

An example

Equations:

$$
\begin{array}{rcll}
0 + n & = & n & (1) \\
(Suc\ m) + n & = & Suc\ (m + n) & (2) \\
(Suc\ m \leq Suc\ n) & = & (m \leq n) & (3) \\
(0 \leq m) & = & True & (4)
\end{array}
$$

Rewriting:

$$
\begin{array}{rcll}
0 + Suc\ 0 & \leq & Suc\ 0 + x & \stackrel{(1)}{=} \\
Suc\ 0 & \leq & Suc\ 0 + x & \stackrel{(2)}{=} \\
Suc\ 0 & \leq & Suc\ (0 + x) &
\end{array}
$$

An example

Equations:

$$\begin{array}{rcll} 0 + n & = & n & (1) \\ (Suc\ m) + n & = & Suc\ (m + n) & (2) \\ (Suc\ m \leq Suc\ n) & = & (m \leq n) & (3) \\ (0 \leq m) & = & True & (4) \end{array}$$

Rewriting:

$$\begin{array}{rcll} 0 + Suc\ 0 & \leq & Suc\ 0 + x & \stackrel{(1)}{=} \\ Suc\ 0 & \leq & Suc\ 0 + x & \stackrel{(2)}{=} \\ Suc\ 0 & \leq & Suc\ (0 + x) & \stackrel{(3)}{=} \\ 0 & \leq & 0 + x & \end{array}$$

An example

Equations:

$$\begin{array}{rcll} 0 + n & = & n & (1) \\ (Suc\ m) + n & = & Suc\ (m + n) & (2) \\ (Suc\ m \leq Suc\ n) & = & (m \leq n) & (3) \\ (0 \leq m) & = & True & (4) \end{array}$$

Rewriting:

$$\begin{array}{rcll} 0 + Suc\ 0 & \leq & Suc\ 0 + x & \stackrel{(1)}{=} \\ Suc\ 0 & \leq & Suc\ 0 + x & \stackrel{(2)}{=} \\ Suc\ 0 & \leq & Suc\ (0 + x) & \stackrel{(3)}{=} \\ 0 & \leq & 0 + x & \stackrel{(4)}{=} \\ & True & & \end{array}$$

More formally

substitution = mapping from variables to terms

More formally

substitution = mapping from variables to terms

- $l = r$ is *applicable* to term $t[s]$
  if there is a substitution σ such that $\sigma(l) = s$

More formally

substitution = mapping from variables to terms

- $l = r$ is *applicable* to term $t[s]$
  if there is a substitution σ such that $\sigma(l) = s$
- Result: $t[\sigma(r)]$

More formally

substitution = mapping from variables to terms

- $l = r$ is *applicable* to term $t[s]$
  if there is a substitution σ such that $\sigma(l) = s$
- Result: $t[\sigma(r)]$
- Note: $t[s] = t[\sigma(r)]$

More formally

substitution = mapping from variables to terms

- $l = r$ is *applicable* to term $t[s]$ if there is a substitution σ such that $\sigma(l) = s$
- Result: $t[\sigma(r)]$
- Note: $t[s] = t[\sigma(r)]$

Example:

Equation: $0 + n = n$

Term: $a + (0 + (b + c))$

More formally

substitution = mapping from variables to terms

- $l = r$ is *applicable* to term $t[s]$ if there is a substitution σ such that $\sigma(l) = s$
- Result: $t[\sigma(r)]$
- Note: $t[s] = t[\sigma(r)]$

Example:

Equation: $0 + n = n$

Term: $a + (0 + (b + c))$

$\sigma = \{n \mapsto b + c\}$

More formally

substitution = mapping from variables to terms

- $l = r$ is *applicable* to term $t[s]$
  if there is a substitution σ such that $\sigma(l) = s$
- Result: $t[\sigma(r)]$
- Note: $t[s] = t[\sigma(r)]$

Example:

Equation: $0 + n = n$

Term: $a + (0 + (b + c))$

$\sigma = \{n \mapsto b + c\}$

Result: $a + (b + c)$

Extension: conditional rewriting

Rewrite rules can be conditional:

$$[\![P_1 \dots P_n]\!] \Longrightarrow l = r$$

Extension: conditional rewriting

Rewrite rules can be conditional:

$$[\![P_1 \dots P_n]\!] \Longrightarrow l = r$$

is *applicable* to term $t[s]$ with σ if

- $\sigma(l) = s$ and
- $\sigma(P_1), \dots, \sigma(P_n)$ are provable (again by rewriting).

Interlude: Variables in Isabelle

Schematic variables

Three kinds of variables:

- bound: $\forall x.\ x = x$
- free: $x = x$

Schematic variables

Three kinds of variables:

- bound: $\forall x.\ x = x$
- free: $x = x$
- schematic: $?x = ?x$ ("unknown")

Schematic variables

Three kinds of variables:

- bound: $\forall x.\ x = x$
- free: $x = x$
- schematic: $?x = ?x$ (“unknown”)

Schematic variables:

Schematic variables

Three kinds of variables:

- bound: $\forall x.\ x = x$
- free: $x = x$
- schematic: $?x = ?x$ (“unknown”)

Schematic variables:

- Logically: free = schematic

Schematic variables

Three kinds of variables:

- bound: $\forall x.\ x = x$
- free: $x = x$
- schematic: $?x = ?x$ ("unknown")

Schematic variables:

- Logically: free = schematic
- Operationally:
  - free variables are fixed
  - schematic variables are instantiated by substitutions

From x to ?x

State lemmas with free variables:

lemma *app_Nil2[simp]: xs @ [] = xs*

From x to ?x

State lemmas with free variables:

lemma *app_Nil2[simp]: xs @ [] = xs*

⋮

done

From x to ?x

State lemmas with free variables:

lemma *app_Nil2[simp]: xs @ [] = xs*

⋮

done

After the proof: Isabelle changes *xs* to *?xs* (internally):

?xs @ [] = ?xs

Now usable with arbitrary values for *?xs*

From x to ?x

State lemmas with free variables:

lemma *app_Nil2[simp]: xs @ [] = xs*

⋮

done

After the proof: Isabelle changes *xs* to *?xs* (internally):

?xs @ [] = ?xs

Now usable with arbitrary values for *?xs*

Example: rewriting

rev(a @ []) = rev a

using *app_Nil2* with $\sigma = \{?xs \mapsto a\}$

Term rewriting in Isabelle

Basic simplification

Goal: *1.* $[\![P_1; \ldots ; P_m]\!] \Longrightarrow C$

apply(*simp add:* $eq_1 \ldots eq_n$)

Basic simplification

Goal: *1.* $[\![P_1; \ldots ; P_m]\!] \Longrightarrow C$

apply(*simp add:* $eq_1 \ldots eq_n$)

Simplify $P_1 \ldots P_m$ and C using

- lemmas with attribute *simp*

Basic simplification

Goal: *1.* $\llbracket P_1; \ldots ; P_m \rrbracket \Longrightarrow C$

apply(*simp add:* $eq_1 \ldots eq_n$)

Simplify $P_1 \ldots P_m$ and C using

- lemmas with attribute *simp*
- rules from **primrec**, **fun** and **datatype**

Basic simplification

Goal: *1.* $[\![P_1; \ldots ; P_m]\!] \Longrightarrow C$

apply(*simp add:* $eq_1 \ldots eq_n$)

Simplify $P_1 \ldots P_m$ and C using

- lemmas with attribute *simp*
- rules from **primrec**, **fun** and **datatype**
- additional lemmas $eq_1 \ldots eq_n$

Basic simplification

Goal: *1.* $[\![P_1; \ldots ; P_m]\!] \Longrightarrow C$

apply(*simp add:* $eq_1 \ldots eq_n$)

Simplify $P_1 \ldots P_m$ and C using

- lemmas with attribute *simp*
- rules from **primrec**, **fun** and **datatype**
- additional lemmas $eq_1 \ldots eq_n$
- assumptions $P_1 \ldots P_m$

Basic simplification

Goal: *1.* $[\![P_1; \ldots ; P_m]\!] \Longrightarrow C$

apply(*simp add:* $eq_1 \ldots eq_n$)

Simplify $P_1 \ldots P_m$ and C using

- lemmas with attribute *simp*
- rules from **primrec**, **fun** and **datatype**
- additional lemmas $eq_1 \ldots eq_n$
- assumptions $P_1 \ldots P_m$

Variations:

- (*simp* ... *del:* ...) removes *simp*-lemmas
- *add* and *del* are optional

auto versus *simp*

- *auto* acts on all subgoals
- *simp* acts only on subgoal 1
- *auto* applies *simp* and more

Termination

Simplification may not terminate.
Isabelle uses *simp*-rules (almost) blindly from left to right.

Termination

Simplification may not terminate.
Isabelle uses *simp*-rules (almost) blindly from left to right.

Example: $f(x) = g(x),\ g(x) = f(x)$

Termination

Simplification may not terminate.
Isabelle uses *simp*-rules (almost) blindly from left to right.

Example: $f(x) = g(x),\ g(x) = f(x)$

$$[\![P_1 \dots P_n]\!] \Longrightarrow l = r$$

is suitable as a *simp*-rule only
if l is “bigger” than r and each P_i

Termination

Simplification may not terminate.
Isabelle uses *simp*-rules (almost) blindly from left to right.

Example: $f(x) = g(x),\ g(x) = f(x)$

$$[\![P_1 \dots P_n]\!] \Longrightarrow l = r$$

is suitable as a *simp*-rule only
if l is “bigger” than r and each P_i

$$n < m \Longrightarrow (n < Suc\ m) = True$$
$$Suc\ n < m \Longrightarrow (n < m) = True$$

Termination

Simplification may not terminate.
Isabelle uses *simp*-rules (almost) blindly from left to right.

Example: $f(x) = g(x),\ g(x) = f(x)$

$$[\![P_1 \dots P_n]\!] \Longrightarrow l = r$$

is suitable as a *simp*-rule only
if l is "bigger" than r and each P_i

n < m $\Longrightarrow$ *(n < Suc m) = True* YES
Suc n < m $\Longrightarrow$ *(n < m) = True* NO

Rewriting with definitions

Definitions do not have the *simp* attribute.

Rewriting with definitions

Definitions do not have the *simp* attribute.

They must be used explicitly: *(simp add: f_def ...)*

Extensions of rewriting

Local assumptions

Simplification of $A \longrightarrow B$:

1. Simplify A to A'
2. Simplify B using A'

Case splitting with simp

$$P(\textit{if } A \textit{ then } s \textit{ else } t)$$
$$=$$
$$(A \longrightarrow P(s)) \wedge (\neg A \longrightarrow P(t))$$

Case splitting with simp

$$P(\textit{if } A \textit{ then } s \textit{ else } t) \\ = \\ (A \longrightarrow P(s)) \wedge (\neg A \longrightarrow P(t))$$

Automatic

Case splitting with simp

$$P(\text{if } A \text{ then } s \text{ else } t) \\ = \\ (A \longrightarrow P(s)) \wedge (\neg A \longrightarrow P(t))$$

Automatic

$$P(\text{case } e \text{ of } 0 \Rightarrow a \mid Suc\ n \Rightarrow b) \\ = \\ (e = 0 \longrightarrow P(a)) \wedge (\forall n.\ e = Suc\ n \longrightarrow P(b))$$

Case splitting with simp

$$P(\textit{if } A \textit{ then } s \textit{ else } t)$$
$$=$$
$$(A \longrightarrow P(s)) \wedge (\neg A \longrightarrow P(t))$$

Automatic

$$P(\textit{case } e \textit{ of } 0 \Rightarrow a \mid Suc\ n \Rightarrow b)$$
$$=$$
$$(e = 0 \longrightarrow P(a)) \wedge (\forall n.\ e = Suc\ n \longrightarrow P(b))$$

By hand: *(simp split: nat.split)*

Case splitting with simp

$$P(\text{if } A \text{ then } s \text{ else } t) = (A \longrightarrow P(s)) \wedge (\neg A \longrightarrow P(t))$$

Automatic

$$P(\text{case } e \text{ of } 0 \Rightarrow a \mid Suc\ n \Rightarrow b) = (e = 0 \longrightarrow P(a)) \wedge (\forall n.\ e = Suc\ n \longrightarrow P(b))$$

By hand: *(simp split: nat.split)*

Similar for any datatype t: *t.split*

Ordered rewriting

Problem: $?x + ?y = ?y + ?x$ does not terminate

Ordered rewriting

Problem: $?x + ?y = ?y + ?x$ does not terminate

Solution: permutative *simp*-rules are used only if the term becomes lexicographically smaller.

Ordered rewriting

Problem: $?x + ?y = ?y + ?x$ does not terminate

Solution: permutative *simp*-rules are used only if the term becomes lexicographically smaller.

Example: $b + a \leadsto a + b$ but not $a + b \leadsto b + a$.

Ordered rewriting

Problem: $?x + ?y = ?y + ?x$ does not terminate

Solution: permutative *simp*-rules are used only if the term becomes lexicographically smaller.

Example: $b + a \leadsto a + b$ but not $a + b \leadsto b + a$.

For types *nat*, *int* etc:

- lemmas *add_ac* sort any sum ($+$)
- lemmas *times_ac* sort any product ($*$)

Ordered rewriting

Problem: $?x + ?y = ?y + ?x$ does not terminate

Solution: permutative *simp*-rules are used only if the term becomes lexicographically smaller.

Example: $b + a \rightsquigarrow a + b$ but not $a + b \rightsquigarrow b + a$.

For types *nat*, *int* etc:

- lemmas *add_ac* sort any sum ($+$)
- lemmas *times_ac* sort any product ($*$)

Example: *(simp add: add_ac)* yields

$$(b + c) + a \rightsquigarrow \cdots \rightsquigarrow a + (b + c)$$

Preprocessing

simp-rules are preprocessed (recursively) for maximal simplification power:

$$\begin{array}{rcl} \neg A & \mapsto & A = False \\ A \longrightarrow B & \mapsto & A \Longrightarrow B \\ A \wedge B & \mapsto & A,\ B \\ \forall x.A(x) & \mapsto & A(?x) \\ A & \mapsto & A = True \end{array}$$

Preprocessing

simp-rules are preprocessed (recursively) for maximal simplification power:

$$\begin{array}{rcl} \neg A & \mapsto & A = False \\ A \longrightarrow B & \mapsto & A \Longrightarrow B \\ A \wedge B & \mapsto & A,\ B \\ \forall x.A(x) & \mapsto & A(?x) \\ A & \mapsto & A = True \end{array}$$

Example:

$(p \longrightarrow q \wedge \neg\, r) \wedge s \quad \mapsto$

Preprocessing

simp-rules are preprocessed (recursively) for maximal simplification power:

$$\begin{array}{rcl} \neg A & \mapsto & A = False \\ A \longrightarrow B & \mapsto & A \Longrightarrow B \\ A \wedge B & \mapsto & A,\ B \\ \forall x. A(x) & \mapsto & A(?x) \\ A & \mapsto & A = True \end{array}$$

Example:

$$(p \longrightarrow q \wedge \neg r) \wedge s \quad \mapsto \quad \left\{ \begin{array}{r} p \Longrightarrow q = True \\ p \Longrightarrow r = False \\ s = True \end{array} \right\}$$

When everything else fails: Tracing

Set trace mode on/off in Proof General:

Isabelle → Settings → Trace simplifier

Output in separate `trace` buffer

Case analysis and structural induction

taken from IsabelleTutorial, Sect. 2, Sect. 3.2, Sect. 3.5

»> slidesNipkow:»> Demo: MyDemo,Trees

Slides for Session 3.2, 1-12 (slidesNipkow 87-93)

»>MyDemo, Induction Heuristics

Slides for Session 2, 57-79

»>MyDemo, Fun

Basic heuristics

Theorems about recursive functions are proved by induction

Basic heuristics

Theorems about recursive functions are proved by induction

Induction on argument number i of f
if f is defined by recursion on argument number i

A tail recursive reverse

primrec *itrev* :: *'a list* $\Rightarrow$ *'a list* $\Rightarrow$ *'a list*

A tail recursive reverse

primrec *itrev* :: *'a list* ⇒ *'a list* ⇒ *'a list* **where**

itrev [] *ys* = *ys* |

itrev (*x*#*xs*) *ys* =

A tail recursive reverse

primrec *itrev :: 'a list ⇒ 'a list ⇒ 'a list* **where**
itrev [] ys = ys |
itrev (x#xs) ys = itrev xs (x#ys)

A tail recursive reverse

primrec *itrev* :: *'a list* $\Rightarrow$ *'a list* $\Rightarrow$ *'a list* **where**
itrev [] *ys = ys* |
itrev (x#xs) *ys = itrev xs (x#ys)*

lemma *itrev xs [] = rev xs*

A tail recursive reverse

primrec *itrev :: 'a list* $\Rightarrow$ *'a list* $\Rightarrow$ *'a list* **where**
itrev [] ys = ys |
itrev (x#xs) ys = itrev xs (x#ys)

lemma *itrev xs [] = rev xs*

Why in this direction?

A tail recursive reverse

primrec *itrev :: 'a list ⇒ 'a list ⇒ 'a list* **where**
itrev []   ys = ys |
itrev (x#xs)   ys = itrev xs (x#ys)

lemma *itrev xs [] = rev xs*

Why in this direction?

Because the lhs is “more complex” than the rhs.

Demo

Generalisation

- Replace constants by variables

Generalisation

- Replace constants by variables
- Generalize free variables
  - by $\forall$ in formula
  - by *arbitrary* in induction proof

Proof search automation

taken from IsabelleTutorial, Sect. 5.12, 5.13

Proof automation tries to apply rules either

- to finish the proof of (sub-)goal
- to simplify the subgoals

We call this the success criterion.

Methods for proof automation are different in

- the success criterion
- the rules they use
- the way in which these rule are applied

Simplification applies rewrite rules repeatedly as long as possible. Classical reasoning uses search and backtracking with rules from predicate logic.

General Methods (Tactics)

blast:

- tries to finish proof of (sub-)goal
- classical reasoner

clarify:

- tries to perform obvious proof steps
- classical reasoner (only safe rule, no splitting of (sub-)goal)

safe:

- tries to perform obvious proof steps
- classical reasoner (only safe rule, splitting)

General Methods (Tactics)

clarsimp:

- tries to finish proof of (sub-)goal
- classical reasoner interleaved with simplification (only safe rule, no splitting)

force:

- tries to finish proof of (sub-)goal
- classical reasoner and simplification

auto:

- tries to perform proof steps on all subgoals
- classical reasoner and simplification (splitting)

More proof methods

Forward proof step in backward proof:

- apply rules to assumptions

Forward proofs (Hilbert style proofs):

- directly prove a theorem from proven theorems

Directives/attributes:

- of: instantiates the variables of a rule to a list of terms
- OF: applies a rule to a list of theorems
- THEN: gives a theorem to named rule and returns the conclusion
- simplified: applies the simplifier to a theorem

More proof methods

Forward proof step in backward proof:

- apply rules to assumptions

Forward proofs (Hilbert style proofs):

- directly prove a theorem from proven theorems

Directives/attributes:

- of: instantiates the variables of a rule to a list of terms
- OF: applies a rule to a list of theorems
- THEN: gives a theorem to named rule and returns the conclusion
- simplified: applies the simplifier to a theorem

Forward proofs: OF

$r[OF\ r_1 \ldots r_n]$

Prove assumption 1 of theorem r with theorem r_1, and assumption 2 with theorem r_2, and ...

Forward proofs: OF

$r[OF\ r_1\ \dots\ r_n]$

Prove assumption 1 of theorem r with theorem r_1, and assumption 2 with theorem r_2, and ...

Rule r	$[\![A_1; \dots ; A_m]\!] \Longrightarrow A$
Rule r_1	$[\![B_1; \dots ; B_n]\!] \Longrightarrow B$
Substitution	$\sigma(B) \equiv \sigma(A_1)$
$r[OF\ r_1]$	

Forward proofs: OF

$r[OF\ r_1 \ldots r_n]$

Prove assumption 1 of theorem r with theorem r_1, and assumption 2 with theorem r_2, and ...

Rule r	$[\![A_1; \ldots ; A_m]\!] \Longrightarrow A$
Rule r_1	$[\![B_1; \ldots ; B_n]\!] \Longrightarrow B$
Substitution	$\sigma(B) \equiv \sigma(A_1)$
$r[OF\ r_1]$	$\sigma([\![B_1; \ldots; B_n; A_2; \ldots; A_m]\!] \Longrightarrow A)$

More proof methods

Method insert:

- inserts a theorem as a new assumption into current subgoal

Method subgoal_tac:

- inserts an arbitrary formula F as assumption
- F becomes additional subgoal

»>MyDemo, subgoal_tac

Chapter 5

Sets, Functions, Relations, and Fixpoints

Sets, Functions, Relations

see IHT 6.1, 6.2, 6.3

- Finite Set Notation
- Set Comprehension
- Binding Operators
- Finiteness and Cardinality
- Function update, Range, Injective - Surjective
- Relations, Predicates

Overview

- Set notation
- Inductively defined sets

Set notation

Sets

Sets over type *'a*:

'a set

Sets

Sets over type *'a*:

$$\textit{'a set} \quad = \quad \textit{'a} \Rightarrow \textit{bool}$$

Sets

Sets over type *'a*:

$$\textit{'a set} \quad = \quad \textit{'a} \Rightarrow \textit{bool}$$

- {}, $\{e_1, \ldots, e_n\}$, {x. P x}

Sets

Sets over type *'a*:

$$\textit{'a set} \quad = \quad \textit{'a} \Rightarrow \textit{bool}$$

- $\{\}$, $\{e_1,\ldots,e_n\}$, $\{x.\ P\ x\}$
- $e \in A$, $A \subseteq B$

Sets

Sets over type *'a*:

$$\text{'a set} \quad = \quad \text{'a} \Rightarrow \textit{bool}$$

- $\{\}$, $\{e_1,\ldots,e_n\}$, $\{x.\ P\ x\}$
- $e \in A$, $A \subseteq B$
- $A \cup B$, $A \cap B$, $A - B$, $- A$

Sets

Sets over type *'a*:

$$\text{'a set} \quad = \quad \text{'a} \Rightarrow bool$$

- $\{\}$, $\{e_1, \ldots, e_n\}$, $\{x.\ P\ x\}$
- $e \in A$, $A \subseteq B$
- $A \cup B$, $A \cap B$, $A - B$, $-A$
- $\bigcup_{x \in A} B\ x$, $\bigcap_{x \in A} B\ x$

Sets

Sets over type *'a*:

$$\textit{'a set} \quad = \quad \textit{'a} \Rightarrow \textit{bool}$$

- $\{\}$, $\{e_1,\ldots,e_n\}$, $\{x.\ P\ x\}$
- $e \in A$, $A \subseteq B$
- $A \cup B$, $A \cap B$, $A - B$, $-A$
- $\bigcup_{x \in A} B\ x$, $\bigcap_{x \in A} B\ x$
- $\{i..j\}$

Sets

Sets over type *'a*:

$$\text{'a set} \quad = \quad \text{'a} \Rightarrow \textit{bool}$$

- $\{\}$, $\{e_1,\ldots,e_n\}$, $\{x.\ P\ x\}$
- $e \in A$, $A \subseteq B$
- $A \cup B$, $A \cap B$, $A - B$, $- A$
- $\bigcup_{x \in A} B\ x$, $\bigcap_{x \in A} B\ x$
- $\{i..j\}$
- *insert* :: 'a $\Rightarrow$ 'a set $\Rightarrow$ 'a set

Sets

Sets over type *'a*:

$$\text{'a set} \quad = \quad \text{'a} \Rightarrow \textit{bool}$$

- $\{\}$, $\{e_1,\ldots,e_n\}$, $\{x.\ P\ x\}$
- $e \in A$, $A \subseteq B$
- $A \cup B$, $A \cap B$, $A - B$, $-A$
- $\bigcup_{x \in A} B\ x$, $\bigcap_{x \in A} B\ x$
- $\{i..j\}$
- *insert* :: 'a ⇒ 'a set ⇒ 'a set
- …

Proofs about sets

Natural deduction proofs:

- `equalityI`: $[\![A \subseteq B; B \subseteq A]\!] \Longrightarrow A = B$

Proofs about sets

Natural deduction proofs:

- `equalityI`: $[\![A \subseteq B; B \subseteq A]\!] \Longrightarrow A = B$
- `subsetI`: $(\bigwedge x.\ x \in A \Longrightarrow x \in B) \Longrightarrow A \subseteq B$

Proofs about sets

Natural deduction proofs:

- `equalityI`: $[\![A \subseteq B; B \subseteq A]\!] \Longrightarrow A = B$
- `subsetI`: $(\bigwedge x.\ x \in A \Longrightarrow x \in B) \Longrightarrow A \subseteq B$
- ... (see Tutorial)

Demo: proofs about sets

Bounded quantifiers

- $\forall x \in A.\ P\ x$

Bounded quantifiers

- $\forall x \in A.\ P\ x \quad \equiv \quad \forall x.\ x \in A \longrightarrow P\ x$

Bounded quantifiers

- $\forall x \in A.\ P\ x \quad \equiv \quad \forall x.\ x \in A \longrightarrow P\ x$
- $\exists x \in A.\ P\ x$

Bounded quantifiers

- $\forall x{\in}A.\ P\ x \quad \equiv \quad \forall x.\ x{\in}A \longrightarrow P\ x$
- $\exists x{\in}A.\ P\ x \quad \equiv \quad \exists x.\ x{\in}A \wedge P\ x$

Bounded quantifiers

- $\forall x{\in}A.\ P\ x \quad \equiv \quad \forall x.\ x{\in}A \longrightarrow P\ x$
- $\exists x{\in}A.\ P\ x \quad \equiv \quad \exists x.\ x{\in}A \wedge P\ x$
- `ballI`: $(\bigwedge x.\ x \in A \Longrightarrow P\ x) \Longrightarrow \forall x{\in}A.\ P\ x$
- `bspec`: $[\![\forall x{\in}A.\ P\ x;\ x \in A]\!] \Longrightarrow P\ x$

Bounded quantifiers

- $\forall x{\in}A.\ P\ x \quad \equiv \quad \forall x.\ x{\in}A \longrightarrow P\ x$
- $\exists x{\in}A.\ P\ x \quad \equiv \quad \exists x.\ x{\in}A \wedge P\ x$
- `ballI`: $(\bigwedge x.\ x \in A \Longrightarrow P\ x) \Longrightarrow \forall x{\in}A.\ P\ x$
- `bspec`: $[\![\forall x{\in}A.\ P\ x;\ x \in A]\!] \Longrightarrow P\ x$
- `bexI`: $[\![P\ x;\ x \in A]\!] \Longrightarrow \exists x{\in}A.\ P\ x$
- `bexE`: $[\![\exists x{\in}A.\ P\ x;\ \bigwedge x.\ [\![x \in A;\ P\ x]\!] \Longrightarrow Q]\!] \Longrightarrow Q$

Inductively defined sets

Example: even numbers

Informally:

Example: even numbers

Informally:

- 0 is even

Example: even numbers

Informally:

- 0 is even
- If n is even, so is $n + 2$

Example: even numbers

Informally:

- 0 is even
- If n is even, so is $n + 2$
- These are the only even numbers

Example: even numbers

Informally:

- 0 is even
- If n is even, so is $n + 2$
- These are the only even numbers

In Isabelle/HOL:

inductive_set *Ev :: nat set* — The set of all even numbers

Example: even numbers

Informally:

- 0 is even
- If n is even, so is $n + 2$
- These are the only even numbers

In Isabelle/HOL:

inductive_set *Ev :: nat set* — The set of all even numbers
where
0 $\in$ *Ev* |
n $\in$ *Ev* $\Longrightarrow$ *n + 2* $\in$ *Ev*

Format of inductive definitions

inductive_set $S :: \tau$ *set*

Format of inductive definitions

inductive_set $S :: \tau$ *set*

where

$\llbracket\, a_1 \in S; \ldots ; a_n \in S; A_1; \ldots; A_k \,\rrbracket \Longrightarrow a \in S \quad |$

$\vdots$

Format of inductive definitions

inductive_set $S :: \tau$ *set*

where

$\llbracket\, a_1 \in S; \ldots ; a_n \in S; A_1; \ldots; A_k \,\rrbracket \Longrightarrow a \in S \quad |$

$\vdots$

where $A_1; \ldots; A_k$ are side conditions not involving S.

Proving properties of even numbers

Easy: $4 \in Ev$

$$0 \in Ev \Longrightarrow 2 \in Ev \Longrightarrow 4 \in Ev$$

Proving properties of even numbers

Easy: $4 \in Ev$

$$0 \in Ev \Longrightarrow 2 \in Ev \Longrightarrow 4 \in Ev$$

Trickier: $m \in Ev \Longrightarrow m{+}m \in Ev$

Proving properties of even numbers

Easy: $4 \in Ev$

$$0 \in Ev \Longrightarrow 2 \in Ev \Longrightarrow 4 \in Ev$$

Trickier: $m \in Ev \Longrightarrow m{+}m \in Ev$

Idea: induction on the length of the derivation of $m \in Ev$

Proving properties of even numbers

Easy: $4 \in Ev$

$$0 \in Ev \Longrightarrow 2 \in Ev \Longrightarrow 4 \in Ev$$

Trickier: $m \in Ev \Longrightarrow m{+}m \in Ev$

Idea: induction on the length of the derivation of $m \in Ev$

Better: induction on the *structure* of the derivation

Proving properties of even numbers

Easy: $4 \in Ev$

$$0 \in Ev \Longrightarrow 2 \in Ev \Longrightarrow 4 \in Ev$$

Trickier: $m \in Ev \Longrightarrow m{+}m \in Ev$

Idea: induction on the length of the derivation of $m \in Ev$

Better: induction on the *structure* of the derivation

Two cases: $m \in Ev$ is proved by

- rule $0 \in Ev$

Proving properties of even numbers

Easy: $4 \in Ev$

$$0 \in Ev \Longrightarrow 2 \in Ev \Longrightarrow 4 \in Ev$$

Trickier: $m \in Ev \Longrightarrow m{+}m \in Ev$

Idea: induction on the length of the derivation of $m \in Ev$

Better: induction on the *structure* of the derivation

Two cases: $m \in Ev$ is proved by

- rule $0 \in Ev$
  $\Longrightarrow m = 0 \Longrightarrow 0{+}0 \in Ev$

Proving properties of even numbers

Easy: $4 \in Ev$

$$0 \in Ev \Longrightarrow 2 \in Ev \Longrightarrow 4 \in Ev$$

Trickier: $m \in Ev \Longrightarrow m{+}m \in Ev$

Idea: induction on the length of the derivation of $m \in Ev$

Better: induction on the *structure* of the derivation

Two cases: $m \in Ev$ is proved by

- rule $0 \in Ev$
  $\Longrightarrow m = 0 \Longrightarrow 0{+}0 \in Ev$
- rule $n \in Ev \Longrightarrow n{+}2 \in Ev$

Proving properties of even numbers

Easy: $4 \in Ev$

$$0 \in Ev \Longrightarrow 2 \in Ev \Longrightarrow 4 \in Ev$$

Trickier: $m \in Ev \Longrightarrow m{+}m \in Ev$

Idea: induction on the length of the derivation of $m \in Ev$

Better: induction on the *structure* of the derivation

Two cases: $m \in Ev$ is proved by

- rule $0 \in Ev$
  $\Longrightarrow m = 0 \Longrightarrow 0{+}0 \in Ev$
- rule $n \in Ev \Longrightarrow n{+}2 \in Ev$
  $\Longrightarrow m = n{+}2$ and $n{+}n \in Ev$ (ind. hyp.!)

Proving properties of even numbers

Easy: $4 \in Ev$

$$0 \in Ev \Longrightarrow 2 \in Ev \Longrightarrow 4 \in Ev$$

Trickier: $m \in Ev \Longrightarrow m{+}m \in Ev$

Idea: induction on the length of the derivation of $m \in Ev$

Better: induction on the *structure* of the derivation

Two cases: $m \in Ev$ is proved by

- rule $0 \in Ev$
  $\Longrightarrow m = 0 \Longrightarrow 0{+}0 \in Ev$
- rule $n \in Ev \Longrightarrow n{+}2 \in Ev$
  $\Longrightarrow m = n{+}2$ and $n{+}n \in Ev$ (ind. hyp.!)
  $\Longrightarrow m{+}m = (n{+}2){+}(n{+}2) = ((n{+}n){+}2){+}2 \in Ev$

Rule induction for Ev

To prove

$$n \in Ev \Longrightarrow P\ n$$

by *rule induction* on $n \in Ev$ we must prove

Rule induction for Ev

To prove

$$n \in Ev \Longrightarrow P\ n$$

by *rule induction* on $n \in Ev$ we must prove

- $P\ 0$

Rule induction for Ev

To prove

$$n \in Ev \Longrightarrow P\ n$$

by *rule induction* on $n \in Ev$ we must prove

- $P\ 0$
- $P\ n \Longrightarrow P(n+2)$

Rule induction for Ev

To prove

$$n \in Ev \Longrightarrow P\ n$$

by *rule induction* on $n \in Ev$ we must prove

- $P\ 0$
- $P\ n \Longrightarrow P(n+2)$

Rule `Ev.induct`:

$$[\![\ n \in Ev;\ P\ 0;\ \bigwedge n.\ P\ n \Longrightarrow P(n+2)\ ]\!] \Longrightarrow P\ n$$

Rule induction in general

Set S is defined inductively.

Rule induction in general

Set S is defined inductively.
To prove

$$x \in S \Longrightarrow P\, x$$

by *rule induction* on $x \in S$

Rule induction in general

Set S is defined inductively.
To prove

$$x \in S \Longrightarrow P\,x$$

by *rule induction* on $x \in S$
we must prove for every rule

$$[\![\, a_1 \in S; \; \ldots \; ; a_n \in S \,]\!] \Longrightarrow a \in S$$

that P is preserved:

Rule induction in general

Set S is defined inductively.
To prove

$$x \in S \Longrightarrow P\,x$$

by *rule induction* on $x \in S$
we must prove for every rule

$$[\![\, a_1 \in S; \ldots ; a_n \in S \,]\!] \Longrightarrow a \in S$$

that P is preserved:

$$[\![\, P\,a_1; \ldots ; P\,a_n \,]\!] \Longrightarrow P\,a$$

Rule induction in general

Set S is defined inductively.
To prove

$$x \in S \Longrightarrow P\,x$$

by *rule induction* on $x \in S$
we must prove for every rule

$$[\![\, a_1 \in S; \ldots ; a_n \in S \,]\!] \Longrightarrow a \in S$$

that P is preserved:

$$[\![\, P\,a_1; \ldots ; P\,a_n \,]\!] \Longrightarrow P\,a$$

In Isabelle/HOL:

apply*(induct rule: S.induct)*

Demo: inductively defined sets

Inductive predicates

$$x \in S \quad \leadsto \quad S\,x$$

Inductive predicates

$$x \in S \quad \rightsquigarrow \quad S\,x$$

Example:

inductive $Ev :: nat \Rightarrow bool$

where

$Ev\ 0 \quad |$

$Ev\ n \Longrightarrow Ev\ (n+2)$

Inductive predicates

$$x \in S \quad \rightsquigarrow \quad S\,x$$

Example:

inductive $Ev :: nat \Rightarrow bool$
where
$Ev\ 0 \quad |$
$Ev\ n \Longrightarrow Ev\ (n + 2)$

Comparison:

predicate: simpler syntax
set: direct usage of $\cup$ etc

Inductive predicates

$$x \in S \quad \rightsquigarrow \quad S\,x$$

Example:

inductive *Ev :: nat ⇒ bool*
where
Ev 0 |
Ev n ⟹ Ev (n + 2)

Comparison:

predicate: simpler syntax
set: direct usage of ∪ etc

Inductive predicates can be of type $\tau_1 \Rightarrow ... \Rightarrow \tau_n \Rightarrow$ *bool*

Automating it

simp and auto

simp rewriting and a bit of arithmetic

auto rewriting and a bit of arithmetic, logic & sets

simp and auto

simp rewriting and a bit of arithmetic

auto rewriting and a bit of arithmetic, logic & sets

- Show you where they got stuck

simp and auto

simp rewriting and a bit of arithmetic

auto rewriting and a bit of arithmetic, logic & sets

- Show you where they got stuck
- highly incomplete wrt logic

blast

- A complete (for FOL) tableaux calculus implementation

blast

- A complete (for FOL) tableaux calculus implementation
- Covers logic, sets, relations, . . .

blast

- A complete (for FOL) tableaux calculus implementation
- Covers logic, sets, relations, ...
- Extensible with intro/elim rules

blast

- A complete (for FOL) tableaux calculus implementation
- Covers logic, sets, relations, ...
- Extensible with intro/elim rules
- Almost no "="

Demo: blast

Well founded relations

see IHT 6.4

- Well founded orderings: Induction
- Complete Lattices Fixpoints
- Knaster-Tarski Theorem

Fixpoints

Importance

- Inductive definitions of sets and relations
- Reminder: relations are sets in Isabelle/HOL
- E.g.: $0 \in$ even
- n $\in$ even ==> n+2 $\in$ even

Properties of Orderings and Functions

Definition 5.1. *Monotone Function*
Let D be a set with an ordering relation $\leq$. A function $f : D \to D$ is called monotone, if $x \leq y \longrightarrow f(x) \leq f(y)$

Remark

The inductive definition above induces a monotone function on sets with the subset relation as ordering:

- f_even :: nat set -> nat set
- f_even $(A) = A \cup \{0\} \cup \{n+2 | n \in A\}$
-
-

Well-founded Orderings

- Partial-order $\leq \subseteq X \times X$ well-founded iff

$$(\forall Y \subseteq X : Y \neq \emptyset \rightarrow (\exists y \in Y : y \text{ minimal in } Y \text{ in respect of } \leq))$$

- Quasi-order $\lesssim$ well-founded iff strict part of $\lesssim$ is well-founded.
- Initial segment: $Y \subseteq X$, left-closed i.e.

$$(\forall y \in Y : (\forall x \in X : x \lesssim y \rightarrow x \in Y))$$

- Initial section of x: $sec(x) = \{y : y < x\}$

Supremum

- Let $(X, \leq)$ be a partial-order and $Y \subseteq X$
- $S \subseteq X$ is a chain iff elements of S are linearly ordered through $\leq$.
- y is an upper bound of Y iff

$$\forall y' \in Y : y' \leq y$$

- Supremum: y is a supremum of Y iff y is an upper bound of Y and

$$\forall y' \in X : ((y' \text{ upper bound of } Y) \rightarrow y \leq y')$$

- Analog: lower bound, Infimum $\inf(Y)$

CPO

- A Partial-order $(D, \sqsubseteq)$ is a complete partial ordering (CPO) iff
  - $\exists$ the smallest element $\perp$ of D (with respect of $\sqsubseteq$)
  - Each chain S has a supremum $\sup(S)$.

Example

Example 5.2. .

- $(\mathcal{P}(X), \subseteq)$ is CPO.
- $(D, \sqsubseteq)$ is CPO with
  - $D = X \nrightarrow Y$: set of all the partial functions f with $\mathrm{dom}(f) \subseteq X$ and $\mathrm{cod}(f) \subseteq Y$.
  - Let $f, g \in X \nrightarrow Y$.

$$f \sqsubseteq g \text{ iff } \mathrm{dom}(f) \subseteq \mathrm{dom}(g) \wedge (\forall x \in \mathrm{dom}(f) : f(x) = g(x))$$

Monotonous, continuous

- $(D, \sqsubseteq)$, $(E, \sqsubseteq')$ CPOs
- $f : D \to E$ monotonous iff

$$(\forall d, d' \in D : d \sqsubseteq d' \to f(d) \sqsubseteq' f(d'))$$

- $f : D \to E$ continuous iff f monotonous and

$$(\forall S \subseteq D : S \text{ chain } \to f(\sup(S)) = \sup(f(S)))$$

- $X \subseteq D$ is admissible iff

$$(\forall S \subseteq X : S \text{ chain } \to \sup(S) \in X)$$

Fixpoint

- $(D, \sqsubseteq)$ CPO, $f : D \to D$
- $d \in D$ fixpoint of f iff
$$f(d) = d$$
- $d \in D$ smallest fixpoint of f iff d fixpoint of f and
$$(\forall d' \in D : d' \text{ fixpoint } \to d \sqsubseteq d')$$

Fixpoint-Theorem

Theorem 5.3 (Fixpoint-Theorem:). *$(D, \sqsubseteq)$ CPO, $f : D \to D$ continuous, then f has a smallest fixpoint μf and*

$$\mu f = \sup\{f^i(\bot) : i \in \mathbb{N}\}$$

Proof: (Sketch)

- $\sup\{f^i(\bot) : i \in \mathbb{N}\}$ fixpoint:

$$\begin{aligned} f(\sup\{f^i(\bot) : i \in \mathbb{N}\}) &= \sup\{f^{i+1}(\bot) : i \in \mathbb{N}\} \\ & \quad \text{(continuous)} \\ &= \sup\{\sup\{f^{i+1}(\bot) : i \in \mathbb{N}\}, \bot\} \\ &= \sup\{f^i(\bot) : i \in \mathbb{N}\} \end{aligned}$$

Fixpoint-Theorem (Cont.)

Fixpoint-Theorem: $(D, \sqsubseteq)$ CPO, $f : D \to D$ continuous, then f has a smallest fixpoint μf and

$$\mu f = \sup\{f^i(\bot) : i \in \mathbb{N}\}$$

Proof: (Continuation)

- $\sup\{f^i(\bot) : i \in \mathbb{N}\}$ smallest fixpoint:
  1. d' fixpoint of f
  2. $\bot \sqsubseteq d'$
  3. f monotonous, d' FP: $f(\bot) \sqsubseteq f(d') = d'$
  4. Induction: $\forall i \in \mathbb{N} : f^i(\bot) \sqsubseteq f^i(d') = d'$
  5. $\sup\{f^i(\bot) : i \in \mathbb{N}\} \sqsubseteq d'$

Induction over $\mathbb{N}$

Induction's principle:

$$(\forall X \subseteq \mathbb{N} : ((0 \in X \wedge (\forall x \in X : x \in X \rightarrow x + 1 \in X))) \rightarrow X = \mathbb{N})$$

Correctness:

1. Let's assume no, so $\exists X \subseteq \mathbb{N} : \mathbb{N} \setminus X \neq \emptyset$
2. Let y be minimum in $\mathbb{N} \setminus X$ (with respect to $<$).
3. $y \neq 0$
4. $y - 1 \in X \wedge y \notin X$
5. Contradiction

Induction over $\mathbb{N}$ (Alternative)

Induction's principle:

$$(\forall X \subseteq \mathbb{N} : (\forall x \in \mathbb{N} : \mathrm{sec}(x) \subseteq X \rightarrow x \in X) \rightarrow X = \mathbb{N})$$

Correctness:

1. Let's assume no, so $\exists X \subseteq \mathbb{N} : \mathbb{N} \setminus X \neq \emptyset$
2. Let y be minimum in $\mathbb{N} \setminus X$ (with respect to $<$).
3. $\mathrm{sec}(y) \subseteq X, y \notin X$
4. Contradiction

Well-founded induction

Induction's principle: Let $(Z, \leq)$ be a well-founded partial order.

$$(\forall X \subseteq Z : (\forall x \in Z : \sec(x) \subseteq X \to x \in X) \to X = Z)$$

Correctness:

1. Let's assume no, so $Z \setminus X \neq \emptyset$
2. Let z be a minimum in $Z \setminus X$ (in respect of $\leq$).
3. $\sec(z) \subseteq X, z \notin X$
4. Contradiction

FP-Induction: Proving properties of fixpoints

Induction's principle: Let $(D, \sqsubseteq)$ CPO, $f : D \to D$ continuous.

$$(\forall X \subseteq D \text{ admissible} : (\bot \in X \wedge (\forall y : y \in X \to f(y) \in X)) \to \mu f \in X)$$

Correctness: Let $X \subseteq D$ admissible.

$$
\begin{array}{lll}
\mu f \in X & \Leftrightarrow & \sup\{f^i(\bot) : i \in \mathbb{N}\} \in X \quad \text{(FP-theorem)} \\
 & \Leftarrow & \forall i \in \mathbb{N} : f^i(\bot) \in X \quad (X \text{ admissible }) \\
 & \Leftarrow & \bot \in X \wedge (\forall n \in \mathbb{N} : f^n(\bot) \in X \to f(f^n(\bot)) \in X) \\
 & & \text{(Induction } \mathbb{N}) \\
 & \Leftarrow & \bot \in X \wedge (\forall y \in X \to f(y) \in X) \quad \text{(Ass.)}
\end{array}
$$

Problem

Exercise 5.4. Let $(D, \sqsubseteq)$ CPO with

- $X = Y = \mathbb{N}$
- $D = X \nrightarrow Y$: set all partial functions f with $\mathrm{dom}(f) \subseteq X$ and $\mathrm{cod}(f) \subseteq Y$.
- Let $f, g \in X \nrightarrow Y$.

$$f \sqsubseteq g \text{ iff } \mathrm{dom}(f) \subseteq \mathrm{dom}(g) \wedge (\forall x \in \mathrm{dom}(f) : f(x) = g(x))$$

Consider

$$\begin{array}{rccl} F : & D & \rightarrow & \mathcal{P}(\mathbb{N} \times \mathbb{N}) \\ & g & \mapsto & \begin{cases} \{(0,1)\} & g = \emptyset \\ \{(x, x \cdot g(x-1)) : x - 1 \in \mathrm{dom}(g)\} \cup \{(0,1)\} & \text{otherwise} \end{cases} \end{array}$$

Problem

Prove:

1. $\forall g \in D : F(g) \in D$, i.e. $F : D \to D$
2. $F : D \to D$ continuous
3. $\forall n \in \mathbb{N} : \mu F(n) = n!$

Note:

- μF can be understood as the semantics of a function's definition

$$
\begin{array}{l}
\text{function Fac}(n : \mathbb{N}_\perp) : \mathbb{N}_\perp =_{\text{def}} \\
\quad \text{if } n = 0 \text{ then } 1 \\
\quad \text{else } n \cdot \text{Fac}(n - 1)
\end{array}
$$

- Keyword: ' functions' in Isabelle

Problem

Exercise 5.5. Prove: Let $G = (V, E)$ be an infinite directed graph with

- G has finitely many roots (nodes without incoming edges).
- Each node has finite out-degree.
- Each node is reachable from a root.

There exists an infinite path that begins on a root.

Complete Lattices and Existence of Fixpoints

Definition 5.6. *Complete Lattice*
A partially ordered set $(L, \leq)$ is a complete lattice if every subset A of L has both a greatest lower bound (the infimum, also called the meet) and a least upper bound (the supremum, also called the join) in $(L, \leq)$. The meet is denoted by $\bigwedge A$, and the join by $\bigvee A$.

Lemma 5.7. *Complete lattices are non empty.*

Theorem 5.8. *Knaster-Tarski*
Let $(L, \leq)$ be a complete lattice and let $f : L \to L$ be a monotone function. Then the set of fixed points of f in L is also a complete lattice.

Consequence 5.9. *The Knaster-Tarski theorem guarantees the existence of least and greatest fixpoints.*

Proof of the Knaster-Tarski theorem

Reformulation

For a complete lattice $(L, \leq)$ and a monotone function $f : L \to L$ on L, the set of all fixpoints of f is also a complete lattice $(P, \leq)$, with:

- $\bigvee P = \bigvee \{x \in L | x \leq f(x)\}$ as the greatest fixpoint of f
- $\bigwedge P = \bigwedge \{x \in L | f(x) <= x\}$ as the least fixpoint of f

Proof: We begin by showing that P has least and greatest elements.
Let $D = \{y \in L | y \leq f(y)\}$ and $x \in D$. Then, because f is monotone, we have $f(x) \leq f(f(x))$, that is $f(x) \in D$.
Now let $u = \bigvee D$. Then $x \leq u$ and $f(x) \leq f(u)$, so $x \leq f(x) \leq f(u)$. Therefore $f(u)$ is an upper bound of D, but u is the least upper bound, so $u \leq f(u)$, i.e. $u \in D$. Then $f(u) \in D$ (from above) and $f(u) \leq u$ hence $f(u) = u$. Because every fixpoint is in D we have that u is the greatest fixpoint of f.

Proof of the Knaster-Tarski theorem (cont.)

The function f is monotone on the dual (complete) lattice $(L^{op}, \geq)$. As we have just proved, its greatest fixpoint there exists. It is the least one on L, so P has least and greatest elements, or more generally that every monotone function on a complete lattice has least and greatest fixpoints.

If $a \in L$ and $b \in L, a \leq b$, we'll write $[a, b]$ for the closed interval with bounds a and b : $\{x \in L | a \leq x \leq b\}$. The closed intervals are also complete lattices.

It remains to prove that P is complete lattice.

Proof of the Knaster-Tarski theorem (cont.)

Let $W \subset P$ and $w = \bigvee W$. We construct a least upper bound of W in P. (The reasoning for the greatest lower bound is analogue.)
For every $x \in W$, we have $x = f(x) \leq f(w)$, i.e., $f(w)$ is an upper bound of W. Since w is the least upper bound of W, $w \leq f(w)$.
Furthermore, for $y \in [w, \bigvee L]$, we have $w \leq f(w) \leq f(y)$. Thus, $f([w, \bigvee L]) \subset [w, \bigvee L]$, and we can consider f to be a monotone function on the complete lattice $[w, \bigvee L]$. Then,
$v = \bigwedge\{x \in [w, \bigvee L] | f(x) \leq x\}$ is the least fixpoint of f in $[w, \bigvee L]$.
We show that v is the least upper bound of W in P.
a) v is in P.
b) v is an upper bound of W, because $v \in [w, \bigvee L]$, i.e., $w \leq v$.
c) v is least. Let z be another upper bound of W in P. Then,
$w \leq z, z \in [w, \bigvee L], z$ is fixpoint, hence $v \leq z$

Lattices in Isabelle

Monotony and Fixpoints

- mono $f \equiv \forall AB.\ A \leq B \longrightarrow f\ A \leq f\ B$ (mono_def)
- Usually subset relation as ordering
- lfp $f \equiv Inf\{u|\ f\ u \leq u\}$ (lfp_def)
- $mono\ f \Longrightarrow lfp\ f = f\ (lfp\ f)$ (lfp_unfold)
- $[|mono\ ?f; ?f\ (inf\ (lfp\ ?f)\ ?P) \leq\ ?P|] \Longrightarrow lfp?f \leq\ ?P$ (lfp_induct)
- gfp $f \equiv Sup\{u|\ u \leq f\ u\}$ (gfp_def)
- $mono\ f \Longrightarrow gfp\ f = f\ (gfp\ f)$ (gfp_unfold)
- $[|mono\ ?f; ?X\ \leq\ ?f\ (sup\ ?X\ (gfp\ ?f))|] \Longrightarrow ?X \leq\ gfp\ ?f$ (coinduct)

Chapter 6

Verifying Functions

Motivation

Verification

Verifying properties of functions is a fundamental task in SE. Hence it is an aspect of theorem proving. In particular, functions definitions allow to express recursive algorithms. Our focus here is on the definition of:

- terminiation/well-definedness properties
- functional properties, i.e., properties relating input parameters to the result (PR-properties).
- Example: A compiler can be considered as a partial function.

In general:

- specification = model + properties
- or
- specification = model_1 + model_2 + relationship

Conceptual aspects

Here: specification = function definition + PR-properties

Verify:

- well-definedness of function by:
  - often structural induction according to parameter types
  - more general: well-founded ordering on parameter space "show that parameters become smaller"
- PR-properties:
  - often structural induction according to parameter types
  - in general, proof technique depends on properties

Discussion

Verification

- works for the full parameter space (in contrast to testing)
- checks for consistency of models and properties
  - models may not reflect what programmer had in mind
  - properties may not reflect what programmer had in mind
  - proofs can have errors
- uses redundancy to find errors
- helps to improve the descriptions

Discussion (cont.)

Formal verification

- avoids misunderstanding
- allows using tools
- avoids errors in proofs
- $\rightsquigarrow$ Isabelle and others

Case study: greatest common devisor

‘

Case study: greatest common devisor

see Gcd.thy

Case study: Quicksort

Assumptions

Given:

```
datatype mapping = lt | ge

fun eval :: "mapping => universe => universe => bool"
    where
  "eval ge xa ya = not(eval lt ya xa)" |
  "[|eval lt ya xa|] ==>  eval lt xa ya = False"
```

Modeling in Isabelle using type classes!

Case study: Quicksort

Shallow embedding of the algorithm:

Case study: Quicksort (cont.)

```
fun qsplit  ::
 "mapping =>  universe => universe list => universe
    list"
  where

  "qsplit xf xa Nil      =  Nil"|
  "qsplit xf xa (ya#x)   =
   (if eval xf ya xa then ya#qsplit xf xa x
                      else qsplit xf xa x)"

fun qsort :: "universe list => universe list" where
  "qsort Nil       = Nil" |
  "qsort (p # l) =
   qsort (qsplit lt p l) @ p # qsort (qsplit ge p l)"
```

Properties to prove

Well-definedness/termination of qsort (1) and qsplit (2)

```
primrec counts :: "'a list => 'a => nat" where
"counts [] x      = 0"   |
"counts (y#yl) x = counts yl x +(if x=y then 1 else 0)
    "
```

lemma qsort_counts(3): “counts xl = counts (qsort xl)”

```
fun qsorted :: " universe list => bool" where
 "qsorted [] = True"|
 "qsorted [x] = True"|
 "qsorted (a#b#l) = (eval ge b a \and qsorted (b#l))"
```

lemma qsort_sort_prop(4): “qsorted (qsort xl)”

Verification of the properties

Ad 1: qsplit is primitive recursive

Ad 2: Idea: length of parameter decreases

```
Auxiliary lemma qsplit_length:
  "length (qsplit f p l) <= length l"
```

⇝ Proof termination with “length” as measure

Verification of the properties (cont.)

```
Auxiliary lemma counts_concat:
"counts (l @ m) x = (counts l x) + (counts m x)"
```

```
Auxiliary theorem qsplit_lt_ge_count [iff]:
  "count (qsplit lt p l) x + count (qsplit ge p l) x =
       count l x"
```

Prove lemma “qsort_counts” by induction

Property 4

Order lifting to lists

```
primrec qall :: "mapping => universe => universe list
    => bool" where
  "qall f p [] = True"
| "qall f p (h # t) = (eval f h p \and qall f p t)"
```

Property 4 (cont.)

Auxiliary Properties

```
theorem qsplit_splits:
   "qall f p (qsplit f p l)"

lemma qall_concat :
  "qall f p (a @ b) = (qall f p a \and qall f p b)"

theorem qsplit_qall :
  "qall f p l ==> qall f p (qsplit g q l)"

theorem qsort_qall :
  "qall f p l  ==> qall f p (qsort l)"
```

prop(4): “sorted (qsort xl)”

Auxiliary lemmatas

```
lemma qsorted_append :
  "[| qsorted l; qall ge p l|] ==> qsorted (p # l)"
```

```
theorem qsorted_concat :
  "[| qsorted a; qsorted b; qall lt p a; qall ge p b
     |] ==> qsorted (a @ p # b)"
```

»> Generic.QSort.thy

Chapter 7

Application: Inductively Defined Sets

Defining sets inductively: Repetition

SessionSlides6.1 starting slide 23

- Rule induction
- Demo inductively defined sets
- Inductive predicates
- Demo

Transition systems

Definition 7.1. *TS*
A transition system (TS) is a pair (Q,T) consisting of

- *a set Q of states;*
- *a binary relation $T \subset (Q * Q)$, usually called the transition relation*

(Other names: state transition system, unlabeled transition system)

Definition 7.2. *LTS*
A labeled transition system (LTS) over Act is a pair (Q,T) consisting of

- *a set Q of states;*
- *a ternary relation $T \subset (Q * Act * Q)$, usually called the transition relation, transitions written as q1 -l-> q2*

Act is called the set of actions.

Transition systems (cont.)

Remark 7.3.

- The action labels express input, output, or an “explanation” of an internal state change.
- Finite automata are LTS.
- Often, transitions systems are equipped with a set of initial states or sets of initial and final states.
- Traces are sequences (qi) of states with $(qi, qi+1) \in T$
- Behavior:: Set of traces beginning at initial states.
- Properties:: expressed in appropriate logic (PDL, CTL ...)

Lemma 7.4. *Every LTS (Q,T) over Act can be expressed by a TS (Q’,T’) such that there is a mapping*
$rep : Q * Act \Rightarrow Q'$
with $q1 - l- > q2 \in T \iff \exists l' : (rep(q1, l'), rep(q2, l)) \in T'$

Proof: <exercise>

Modeling: Case study Elevator

Model of an elevator control system: Description

- Design the logic to move one lift between 3 floors satisfying:
- The lift has for each floor one button which, if pressed, causes the lift to visit that floor. It is cancelled when the lift visits the floor.
- Each floor has a button to request the lift. It is cancelled when the lift visits the floor.
- The lift remains in middle floor if no requests are pending.
- Properties
- All requests for floors from the lift must be serviced eventually.
- All requests from floors must be serviced eventually.

Modeling: Case study Elevator

Datatypes and actions

```
datatype floor = F0 | F1 | F2
(* actions *)
datatype action = Call floor  (* input message *)
                | GoTo floor  (* input message *)
                | Open        (* output message *)
                | Move      (* internal message *)

(* types for elevator state *)
datatype direction = UP | DW
datatype door      = CL | OP
(* elevator state *)
"action * floor * direction * door * (floor set)"
(* where | last move | open/closed | what to serve *)
```

Datatypes and actions: Transition relation

```
inductive_set tr :: "(state * state) set" where
"[|g \<notin> T; \<not> (f = g \<and> d = OP)|] ==>
((a,f,r d,T),(Call g,f,r,d,T \<union> {g})) \<in> tr"|
"[|g \<notin> T; \<not> (f = g \<and> d = OP)|] ==>
((a,f,r,d,T),(GoTo g,f,r,d,T \<union> {g})) \<in> tr"|
"f\<in>T==>((a,f,r,d,T),(Open,f,r,OP,T-{f}))\<in>tr"|
"((a,F1,r,d,{F0}),(Move,F0,DW,CL,{F0})) \<in> tr"|
"((a,F1,r,d,{F2}),(Move,F2,UP,CL,{F2})) \<in> tr"|
"F0\<notin>T==>((a,F0,r,d,T),(Move,F1,UP,CL,T))\<in>tr
"F2\<notin>T==>((a,F2,r,d,T),(Move,F1,DW,CL,T))\<in>tr
"[|F1\<notin>T; F2\<in>T|] ==>
((a,F1,UP,d,T),(Move,F2,UP,CL,T)) \<in> tr"|
"[|F1\<notin>T; F0\<in>T|] ==>
((a,F1,DW,d,T),(Move,F0,DW,CL,T)) \<in> tr"
```

Traces

Defining sets of traces

```
types trace = "nat => state"

coinductive_set traces :: "trace set" where
"[| t \<in> traces; (s, t 0) \<in> tr |]  ==>
(\<lambda>n. case n of 0 => s | Suc x => t x) \<in>
    traces"

(* Functions on traces *)

definition head :: "trace => state" where
  "head t \<equiv> t 0"

definition drp :: "trace => nat => trace" where
  "drp t n \<equiv> (\<lambda>x. t (n + x))"
```

Properties of Traces

Important properties

- lemma [iff]: “drp (drp t n) m = drp t (n + m)”
- lemma drp_traces: “$t \in traces \Longrightarrow drp\ \ t\ \ n \in traces$”

Reasoning about finite transition systems

Logic for expressing properties of traces

- For every floor f: If f is a target floor, the elevator will eventually reach the floor and open the door.
- Always («To f» –> Finally («Op» and «At f»))
- $\leadsto$ Temporal logic. Here e.g. LTL
- Formulae built with Atoms, $\neg, \wedge, \Box, \Diamond$
- Interpretations: Kripke structures (Q, I, T, L)
- A transition relation $T \subseteq Q * Q$ such that $\forall q \in Q.\exists q' \in Q.(q, q') \in T$
- a labeling (or interpretation) function $L : Q \rightarrow 2^{Atoms}$

Reasoning about finite transition systems

Remark 7.5.

- Since T is left-total, it is always possible to construct an infinite path through the Kripke structure. A deadlock state qd can be expressed by single outgoing edge back to qd itself.
- Labeling states (elevator)

```
datatype atom = Up | Op | At floor | To floor

fun L :: "state => atom => bool" where
"L (_, _, UP, _, _) Up = True" |
"L (_, _, DW, _, _) Up = False" |
"L (_, _, _, CL, _) Op = False" |
"L (_, _, _, OP, _) Op = True" |
"L (_, f, _, _, _) (At g) = (f = g)" |
"L (_, _, _, _, fs) (To f) = (f \<in> fs)"
```

Reasoning about finite transition systems (cont.)

- The labeling function L defines for each state q in Q the set L(s) of all atomic propositions that are valid in s.
- Semantics of LTL

```
primrec valid ::
"trace => formula => bool"   ("(_ |= _)" [80, 80]
    80) where
     "t |= Atom a     = ( a \<in> L (head t) )"
   | "t |= Neg f      = ( \<not> (t |= f) )"
   | "t |= And f g    = ( t |= f \<and> t |= g )"
   | "t |= Always f   = ( \<forall>n. drp t n |= f )"
   | "t |= Finally f = ( \<exists>n. drp t n |= f )"
```

- »> Elevator.thy

Chapter 8

Application: Programming Language Semantics

Programming Language Semantics

Software Foundations Book

- Material: http://sct.ethz.ch/teaching/ss2004/sps/lecture.html
- PM intro
- PM bigstep semantics
- Demo MyWhile.thy
- PM smallstep semantics
- Denotational semantics
- Axiomatic semantics: Hoare Logic.
- Demo MyHoare.thy

Why Formal Semantics?

- Programming language design
  - Formal verification of language properties
  - Reveal ambiguities
  - Support for standardization
- Implementation of programming languages
  - Compilers
  - Interpreters
  - Portability
- Reasoning about programs
  - Formal verification of program properties
  - Extended static checking

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Language Properties

- Type safety:
  In each execution state, a variable of type T holds a value of T or a subtype of T
- Very important question for language designers
- Example:
  If String is a subtype of Object, should `String[]` be a subtype of `Object[]`?

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Peter Müller—Semantics of Programming Languages, SS04– p.8

Language Properties

- Type safety:
  In each execution state, a variable of type T holds a value of T or a subtype of T
- Very important question for language designers
- Example:
  If String is a subtype of Object, should `String[]` be a subtype of `Object[]`?

```
void m(Object[] oa) {
  oa[0]=new Integer(5);
}
```

```
String[] sa=new String[10];
m(sa);
String s = sa[0];
```

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Peter Müller—Semantics of Programming Languages, SS04 – p.8

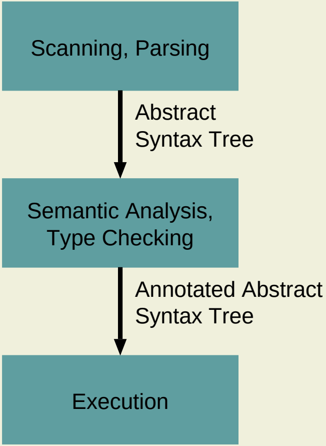
Language Definition

Dynamic Semantics	▸ State of a program execution ▸ Transformation of states
Static Semantics	▸ Type rules ▸ Name resolution
Syntax	▸ Syntax rules, defined by grammar

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Peter Müller—Semantics of Programming Languages, SS04 – p.12

Compilation and Execution



Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Peter Müller—Semantics of Programming Languages, SS04 – p.13

Three Kinds of Semantics

- Operational semantics
  - Describes execution on an **abstract machine**
  - Describes **how** the effect is achieved
- Denotational semantics
  - Programs are regarded as **functions** in a mathematical domain
  - Describes **only the effect**, not how it is obtained
- Axiomatic semantics
  - **Specifies properties** of the effect of executing a program are expressed
  - Some aspects of the computation may be **ignored**

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Peter Müller—Semantics of Programming Languages, SS04 – p.14

Operational Semantics

```
y := 1;
while not(x=1) do ( y := x*y; x := x-1 )
```

- “First we assign 1 to y, then we test whether x is 1 or not. If it is then we stop and otherwise we update y to be the product of x and the previous value of y and then we decrement x by 1. Now we test whether the new value of x is 1 or not. . . ”
- Two kinds of operational semantics
  - Natural Semantics
  - Structural Operational Semantics

ETH Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Peter Müller—Semantics of Programming Languages, SS04 – p.15

Denotational Semantics

```
y := 1;
while not(x=1) do ( y := x*y; x := x-1 )
```

- “The program computes a partial function from states to states: the final state will be equal to the initial state except that the value of x will be 1 and the value of y will be equal to the factorial of the value of x in the initial state”
- Two kinds of denotational semantics
  - Direct Style Semantics
  - Continuation Style Semantics

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Axiomatic Semantics

```
y := 1;
while not(x=1) do ( y := x*y; x := x-1 )
```

- "If x$= n$ holds before the program is executed then y$= n!$ will hold when the execution terminates (if it terminates)"
- Two kinds of axiomatic semantics
  - Partial correctness
  - Total correctness

ETH Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Peter Müller—Semantics of Programming Languages, SS04 – p.17

Abstraction

Concrete language implementation
Operational semantics
Denotational semantics
Axiomatic semantics
Abstract descrption

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Peter Müller—Semantics of Programming Languages, SS04 – p.18

Selection Criteria

- Constructs of the programming language
  - Imperative
  - Functional
  - Concurrent
  - Object-oriented
  - Non-deterministic
  - Etc.
- Application of the semantics
  - Understanding the language
  - Program verification
  - Prototyping
  - Compiler construction
  - Program analysis
  - Etc.

ETH Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Peter Müller—Semantics of Programming Languages, SS04 – p.19

The Language IMP

- Expressions
  - Boolean and arithmetic expressions
  - No side-effects in expressions
- Variables
  - All variables range over integers
  - All variables are initialized
  - No global variables
- IMP does not include
  - Heap allocation and pointers
  - Variable declarations
  - Procedures
  - Concurrency

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Peter Müller—Semantics of Programming Languages, SS04 – p.30

Syntax of IMP: Characters and Tokens

Characters

Letter	= 'A' ... 'Z' | 'a' ... 'z'
Digit	= '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'

Tokens

Ident	= Letter { Letter | Digit }
Integer	= Digit { Digit }
Var	= Ident

Syntax of IMP: Expressions

Arithmetic expressions

Aexp	= Aexp Op Aexp | Var | Integer
Op	= '+' | '-' | '*' | '/' | 'mod'

Boolean expressions

Bexp	= Bexp 'or' Bexp | Bexp 'and' Bexp
	| 'not' Bexp | Aexp RelOp Aexp
RelOp	= '=' | '#' | '<' | '<=' | '>' | '>='

ETH Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Peter Müller—Semantics of Programming Languages, SS04 – p.32

Syntax of IMP: Statemens

Stm	= 'skip'
	| Var ':=' Aexp
	| Stm ';' Stm
	| 'if' Bexp 'then' Stm 'else' Stm 'end'
	| 'while' Bexp 'do' Stm 'end'

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Peter Müller—Semantics of Programming Languages, SS04 – p.33

Notation

Meta-variables (written in *italic* font)

x, y, z	for variables (Var)
e, e', e_1, e_2	for arithmetic expressions (Aexp)
b, b_1, b_2	for boolean expressions (Bexp)
s, s', s_1, s_2	for statements (Stm)

Keywords are written in `typewriter` font

Syntax of IMP: Example

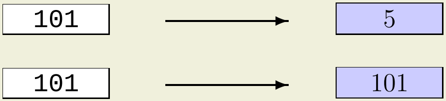
```
res := 1;
while n > 1 do
  res := res * n;
  n := n - 1
end
```

ETH Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Peter Müller—Semantics of Programming Languages, SS04 – p.35

Semantic Categories

Syntactic category: Integer Semantic category: Val $= \mathbb{Z}$



- Semantic functions map elements of syntactic categories to elements of semantic categories
- To define the semantics of IMP, we need semantic functions for
  - Arithmetic expressions (syntactic category Aexp)
  - Boolean expressions (syntactic category Bexp)
  - Statements (syntactic category Stm)

States

x+1 ⟶ ??

- The meaning of an expression depends on the values bound to the variables that occur in it
- A state associates a value to each variable

$$\text{State} : \text{Var} \longrightarrow \text{Val}$$

- We represent a state σ as a finite function

$$\sigma = \{x_1 \mapsto v_1, x_2 \mapsto v_2, \ldots, x_n \mapsto v_n\}$$

where $x_1, x_2, \ldots, x_n$ are different elements of Var and $v_1, v_2, \ldots, v_n$ are elements of Val.

Semantics of Arithmetic Expressions

The semantic function

$$\mathcal{A} : \text{Aexp} \longrightarrow \text{State} \longrightarrow \text{Val}$$

maps an arithmetic expression e and a state σ to a value $\mathcal{A}[\![e]\!]\sigma$

$$\begin{array}{lll} \mathcal{A}[\![x]\!]\sigma & = \sigma(x) & \\ \mathcal{A}[\![i]\!]\sigma & = i & \text{for } i \in \mathbb{Z} \\ \mathcal{A}[\![e_1 \; op \; e_2]\!]\sigma & = \mathcal{A}[\![e_1]\!]\sigma \; \overline{op} \; \mathcal{A}[\![e_2]\!]\sigma & \text{for } op \in \text{Op} \end{array}$$

$\overline{op}$ is the operation $\text{Val} \times \text{Val} \longrightarrow \text{Val}$ corresponding to op

Semantics of Boolean Expressions

The semantic function

$$\mathcal{B} : \text{Bexp} \longrightarrow \text{State} \longrightarrow \text{Bool}$$

maps a boolean expression b and a state σ to a truth value $\mathcal{B}[\![b]\!]\sigma$

$$\mathcal{B}[\![e_1\ op\ e_2]\!]\sigma \quad = \begin{cases} tt & \text{if } \mathcal{A}[\![e_1]\!]\sigma\ \overline{op}\ \mathcal{A}[\![e_2]\!]\sigma \\ ff & \text{otherwise} \end{cases}$$

$op \in \text{RelOp}$ and $\overline{op}$ is the relation $\text{Val} \times \text{Val}$ corresponding to op

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Boolean Expressions (cont'd)

$$\mathcal{B}[\![b_1 \texttt{ or } b_2]\!]\sigma = \begin{cases} tt & \text{if } \mathcal{B}[\![b_1]\!]\sigma = tt \text{ or } \mathcal{B}[\![b_2]\!]\sigma = tt \\ ff & \text{otherwise} \end{cases}$$

$$\mathcal{B}[\![b_1 \texttt{ and } b_2]\!]\sigma = \begin{cases} tt & \text{if } \mathcal{B}[\![b_1]\!]\sigma = tt \text{ and } \mathcal{B}[\![b_2]\!]\sigma = tt \\ ff & \text{otherwise} \end{cases}$$

$$\mathcal{B}[\![\texttt{not } b]\!]\sigma = \begin{cases} tt & \text{if } \mathcal{B}[\![b]\!]\sigma = ff \\ ff & \text{otherwise} \end{cases}$$

Operational Semantics of Statements

- Evaluation of an expression in a state yields a value

```
x + 2 * y
```

$\mathcal{A}$: Aexp $\longrightarrow$ State $\longrightarrow$ Val

- Execution of a statement modifies the state

```
x := 2 * y
```

- Operational semantics describe **how** the state is modified during the execution of a statement

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Peter Müller—Semantics of Programming Languages, SS04 – p.57

Big-Step and Small-Step Semantics

- Big-step semantics describe how the **overall** results of the executions are obtained
  - Natural semantics
- Small-step semantics describe how the **individual steps** of the computations take place
  - Structural operational semantics
  - Abstract state machines

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Peter Müller—Semantics of Programming Languages, SS04 – p.58

Transition Systems

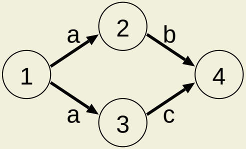
- A transition system is a tuple $(\Gamma, T, \triangleright)$
  - Γ: a set of **configurations**
  - T: a set of **terminal configurations**, $T \subseteq \Gamma$
  - $\triangleright$: a **transition relation**, $\triangleright \subseteq \Gamma \times \Gamma$

- Example: Finite automaton

$$\Gamma = \{\langle w, S\rangle \mid w \in \{a, b, c\}^*, S \in \{1, 2, 3, 4\}\}$$

$$T = \{\langle \epsilon, S\rangle \mid S \in \{1, 2, 3, 4\}\}$$

$$\triangleright = \{(\langle aw, 1\rangle \rightarrow \langle w, 2\rangle), (\langle aw, 1\rangle \rightarrow \langle w, 3\rangle), (\langle bw, 2\rangle \rightarrow \langle w, 4\rangle), (\langle cw, 3\rangle \rightarrow \langle w, 4\rangle)\}$$



Transitions in Natural Semantics

- Two types of configurations for operational semantics
  1. $\langle s, \sigma \rangle$, which represents that the statement s is to be executed in state σ
  2. σ, which represents a terminal state
- The transition relation $\rightarrow$ describes how executions take place
  - Typical transition: $\langle s, \sigma \rangle \rightarrow \sigma'$
  - Example: $\langle \mathtt{skip}, \sigma \rangle \rightarrow \sigma$

$$\Gamma = \{\langle s, \sigma \rangle \mid s \in \text{Stm}, \sigma \in \text{State}\} \cup \text{State}$$
$$T = \text{State}$$
$$\rightarrow \subseteq \{\langle s, \sigma \rangle \mid s \in \text{Stm}, \sigma \in \text{State}\} \times \text{State}$$

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Peter Müller—Semantics of Programming Languages, SS04 – p.61

Rules

- Transition relation is specified by rules

$$\frac{\varphi_1, \ldots, \varphi_n}{\psi} \quad \text{if } \textit{Condition}$$

where $\varphi_1, \ldots, \varphi_n$ and ψ are transitions

- Meaning of the rule

If *Condition* and $\varphi_1, \ldots, \varphi_n$ then ψ

- Terminology
  - $\varphi_1, \ldots, \varphi_n$ are called **premises**
  - ψ is called **conclusion**
  - A rule without premises is called **axiom**

ETH Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Peter Müller—Semantics of Programming Languages, SS04 – p.62

Notation

- Updating States: $\sigma[y \mapsto v]$ is the function that
  - overrides the association of y in σ by $y \mapsto v$ or
  - adds the new association $y \mapsto v$ to σ

$$(\sigma[y \mapsto v])(x) = \begin{cases} v & \text{if } x = y \\ \sigma(x) & \text{if } x \neq y \end{cases}$$

ETH Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Peter Müller—Semantics of Programming Languages, SS04 – p.63

Natural Semantics of IMP

- `skip` does not modify the state

$$\frac{}{\langle \texttt{skip}, \sigma\rangle \rightarrow \sigma}$$

- x:=e assigns the value of e to variable e

$$\frac{}{\langle x \texttt{:=} e, \sigma\rangle \rightarrow \sigma[x \mapsto \mathcal{A}[\![e]\!]\sigma]}$$

- Sequential composition s_1;s_2
  - First, s_1 is executed in state σ, leading to σ'
  - Then s_2 is executed in state σ'

$$\frac{\langle s_1, \sigma\rangle \rightarrow \sigma', \langle s_2, \sigma'\rangle \rightarrow \sigma''}{\langle s_1 \texttt{;} s_2, \sigma\rangle \rightarrow \sigma''}$$

ETH Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Peter Müller—Semantics of Programming Languages, SS04 – p.64

Natural Semantics of IMP (cont'd)

- Conditional statement `if` b `then` s_1 `else` s_2 `end`
  - If b holds, s_1 is executed
  - If b does not hold, s_2 is executed

$$\frac{\langle s_1, \sigma\rangle \rightarrow \sigma'}{\langle \texttt{if}\ b\ \texttt{then}\ s_1\ \texttt{else}\ s_2\ \texttt{end}, \sigma\rangle \rightarrow \sigma'} \quad \text{if } \mathcal{B}[\![b]\!]\sigma = tt$$

$$\frac{\langle s_2, \sigma\rangle \rightarrow \sigma'}{\langle \texttt{if}\ b\ \texttt{then}\ s_1\ \texttt{else}\ s_2\ \texttt{end}, \sigma\rangle \rightarrow \sigma'} \quad \text{if} \mathcal{B}[\![b]\!]\sigma = \mathit{ff}$$

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Natural Semantics of IMP (cont'd)

- Loop statement `while` b `do` s `end`
  - If b holds, s is executed once, leading to state σ'
  - Then the whole while-statement is executed again σ'

$$\frac{\langle s, \sigma\rangle \rightarrow \sigma', \langle \texttt{while}\ b\ \texttt{do}\ s\ \texttt{end}, \sigma'\rangle \rightarrow \sigma''}{\langle \texttt{while}\ b\ \texttt{do}\ s\ \texttt{end}, \sigma\rangle \rightarrow \sigma''} \quad \text{if } \mathcal{B}[\![b]\!]\sigma = tt$$

  - If b does not hold, the while-statement does not modify the state

$$\frac{}{\langle \texttt{while}\ b\ \texttt{do}\ s\ \texttt{end}, \sigma\rangle \rightarrow \sigma} \quad \text{if } \mathcal{B}[\![b]\!]\sigma = f\!f$$

ETH Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Rule Instantiations

- Rules are actually **rule schemes**
  - Meta-variables stand for arbitrary variables, expressions, statements, states, etc.
  - To apply rules, they have to be **instantiated** by selecting particular variables, expressions, statements, states, etc.
- Assignment rule **scheme**

$$\langle x \texttt{:=} e, \sigma\rangle \rightarrow \sigma[x \mapsto \mathcal{A}[\![e]\!]\sigma]$$

- Assignment rule **instance**

$$\langle \texttt{v:=v+1}, \{\texttt{v} \mapsto 3\}\rangle \rightarrow \{\texttt{v} \mapsto 4\}$$

Derivations: Example

- What is the final state if statement

  `z:=x; x:=y; y:=z`

  is executed in state $\{\mathtt{x} \mapsto 5, \mathtt{y} \mapsto 7, \mathtt{z} \mapsto 0\}$ (abbreviated by $[5, 7, 0]$)?

$$\dfrac{\dfrac{\langle \mathtt{z:=x}, [5,7,0]\rangle \rightarrow [5,7,5], \langle \mathtt{x:=y}, [5,7,5]\rangle \rightarrow [7,7,5]}{\langle \mathtt{z:=x;\ x:=y}, [5,7,0]\rangle \rightarrow [7,7,5]}, \quad \langle \mathtt{y:=z}, [7,7,5]\rangle \rightarrow [7,5,5]}{\langle \mathtt{z:=x;\ x:=y;\ y:=z}, [5,7,0]\rangle \rightarrow [7,5,5]}$$

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Derivation Trees

- Rule instances can be combined to derive a transition $\langle s, \sigma \rangle \rightarrow \sigma'$
- The result is a **derivation tree**
  - The root is the transition $\langle s, \sigma \rangle \rightarrow \sigma'$
  - The leaves are axiom instances
  - The internal nodes are conclusions of rule instances and have the corresponding premises as immediate children
- The conditions of all instantiated rules must be satisfied
- There can be several derivations for one transition (non-deterministic semantics)

Termination

- The execution of a statement s in state σ
  - **terminates** iff there is a state σ' such that $\langle s, \sigma \rangle \to \sigma'$
  - **loops** iff there is no state σ' such that $\langle s, \sigma \rangle \to \sigma'$
- A statement s
  - **always terminates** if the execution in a state σ terminates for all choices of σ
  - **always loops** if the execution in a state σ loops for all choices of σ

Semantic Equivalence

- Definition

  Two statements s_1 and s_2 are **semantically equivalent** (denoted by $s_1 \equiv s_2$) if the following property holds for all states σ, σ':

  $$\langle s_1, \sigma \rangle \rightarrow \sigma' \Leftrightarrow \langle s_2, \sigma \rangle \rightarrow \sigma'$$

- Example

  `while` b `do` s `end` $\equiv$
  `if` b `then` s`;` `while` b `do` s `end`

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Structural Operational Semantics

- The emphasis is on the **individual steps** of the execution
  - Execution of assignments
  - Execution of tests
- Describing small steps of the execution allows one to express the **order of execution** of individual steps
  - Interleaving computations
  - Evaluation order for expressions (not shown in the course)
- Describing always the **next small step** allows one to express **properties of looping programs**

ETH Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Peter Müller—Semantics of Programming Languages, SS04 – p.100

Transitions in SOS

- The configurations are the same as for natural semantics
- The transition relation $\rightarrow_1$ can have two forms
- $\langle s, \sigma \rangle \rightarrow_1 \langle s', \sigma' \rangle$: the execution of s from σ is **not completed** and the remaining computation is expressed by the intermediate configuration $\langle s', \sigma' \rangle$
- $\langle s, \sigma \rangle \rightarrow_1 \sigma'$: the execution of s from σ **has terminated** and the final state is σ'
- A transition $\langle s, \sigma \rangle \rightarrow_1 \gamma$ describes the **first step** of the execution of s from σ

ETH Eidgenössische Technische Hochschule Zürich Swiss Federal Institute of Technology Zurich

Peter Müller—Semantics of Programming Languages, SS04 – p.101

Transition System

$$\Gamma = \{\langle s, \sigma \rangle \mid s \in \textsf{Stm}, \sigma \in \textsf{State}\} \cup \textsf{State}$$
$$T = \textsf{State}$$
$$\rightarrow_1 \subseteq \{\langle s, \sigma \rangle \mid s \in \textsf{Stm}, \sigma \in \textsf{State}\} \times \Gamma$$

- We say that $\langle s, \sigma \rangle$ is **stuck** if there is no γ such that $\langle s, \sigma \rangle \rightarrow_1 \gamma$

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

SOS of IMP

- `skip` does not modify the state

$$\langle \texttt{skip}, \sigma \rangle \rightarrow_1 \sigma$$

- x `:=` e assigns the value of e to variable x

$$\langle x\texttt{:=}e, \sigma \rangle \rightarrow_1 \sigma[x \mapsto \mathcal{A}[\![e]\!]\sigma]$$

- `skip` and assignment require only one step
- Rules are analogous to natural semantics

$$\langle \texttt{skip}, \sigma \rangle \rightarrow \sigma$$

$$\langle x\texttt{:=}e, \sigma \rangle \rightarrow \sigma[x \mapsto \mathcal{A}[\![e]\!]\sigma]$$

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Peter Müller—Semantics of Programming Languages, SS04 – p.103

SOS of IMP: Sequential Composition

- Sequential composition $s_1 \,;\, s_2$
- First step of executing $s_1 \,;\, s_2$ is the first step of executing s_1
- s_1 is executed in one step

$$\frac{\langle s_1, \sigma\rangle \rightarrow_1 \sigma'}{\langle s_1 \,;\, s_2, \sigma\rangle \rightarrow_1 \langle s_2, \sigma'\rangle}$$

- s_1 is executed in several steps

$$\frac{\langle s_1, \sigma\rangle \rightarrow_1 \langle s_1', \sigma'\rangle}{\langle s_1 \,;\, s_2, \sigma\rangle \rightarrow_1 \langle s_1' \,;\, s_2, \sigma'\rangle}$$

ETH Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Peter Müller—Semantics of Programming Languages, SS04 – p.104

SOS of IMP: Conditional Statement

- The first step of executing `if` b `then` s_1 `else` s_2 `end` is to determine the outcome of the test and thereby which branch to select

$$\langle \texttt{if}\ b\ \texttt{then}\ s_1\ \texttt{else}\ s_2\ \texttt{end}, \sigma\rangle \rightarrow_1 \langle s_1, \sigma\rangle \quad \text{if } \mathcal{B}[\![b]\!]\sigma = tt$$

$$\langle \texttt{if}\ b\ \texttt{then}\ s_1\ \texttt{else}\ s_2\ \texttt{end}, \sigma\rangle \rightarrow_1 \langle s_2, \sigma\rangle \quad \text{if} \mathcal{B}[\![b]\!]\sigma = f\!f$$

ETH Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Peter Müller—Semantics of Programming Languages, SS04 – p.105

Alternative for Conditional Statement

- The first step of executing `if` b `then` s_1 `else` s_2 `end` is the first step of the branch determined by the outcome of the test

$$\frac{\langle s_1, \sigma \rangle \rightarrow_1 \sigma'}{\langle \texttt{if}\ b\ \texttt{then}\ s_1\ \texttt{else}\ s_2\ \texttt{end}, \sigma \rangle \rightarrow_1 \sigma'} \quad \text{if } \mathcal{B}[\![b]\!]\sigma = tt$$

$$\frac{\langle s_1, \sigma \rangle \rightarrow_1 \langle s_1', \sigma' \rangle}{\langle \texttt{if}\ b\ \texttt{then}\ s_1\ \texttt{else}\ s_2\ \texttt{end}, \sigma \rangle \rightarrow_1 \langle s_1', \sigma' \rangle} \quad \text{if } \mathcal{B}[\![b]\!]\sigma = tt$$

and two similar rules for $\mathcal{B}[\![b]\!]\sigma = f\!f$

- Alternatives are equivalent for IMP
- Choice is important for languages with parallel execution

SOS of IMP: Loop Statement

- The first step is to unrole the loop

$$\langle \texttt{while}\ b\ \texttt{do}\ s\ \texttt{end}, \sigma\rangle \rightarrow_1$$
$$\langle \texttt{if}\ b\ \texttt{then}\ s\texttt{;while}\ b\ \texttt{do}\ s\ \texttt{end}\ \texttt{else}\ \texttt{skip}\ \texttt{end}, \sigma\rangle$$

- Recall that `while` b `do` s `end` and `if` b `then` s`;while` b `do` s `end else skip end` are semantically equivalent in the natural semantics

ETH Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Alternatives for Loop Statement

- The first step is to decide the outcome of the test and thereby whether to unrole the body of the loop or to terminate

$$\langle \texttt{while}\ b\ \texttt{do}\ s\ \texttt{end}, \sigma\rangle \rightarrow_1 \langle s\texttt{;while}\ b\ \texttt{do}\ s\ \texttt{end}, \sigma\rangle \quad \text{if } \mathcal{B}[\![b]\!]\sigma = tt$$

$$\langle \texttt{while}\ b\ \texttt{do}\ s\ \texttt{end}, \sigma\rangle \rightarrow_1 \sigma \quad \text{if } \mathcal{B}[\![b]\!]\sigma = f\!f$$

- Or combine with the alternative semantics of the conditional statement
- Alternatives are equivalent for IMP

ETH Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Peter Müller—Semantics of Programming Languages, SS04 – p.108

Derivation Sequences

- A **derivation sequence** of a statement s starting in state σ is a sequence $\gamma_0, \gamma_1, \gamma_2, \ldots$, where
  - $\gamma_0 = \langle s, \sigma \rangle$
  - $\gamma_i \rightarrow_1 \gamma_{i+1}$ for $0 \leq i$
- A derivation sequence is either **finite** or **infinite**
  - Finite derivation sequences end with a configuration that is either a terminal configuration or a stuck configuration
- Notation
  - $\gamma_0 \rightarrow_1^i \gamma_i$ indicates that there are i steps in the execution from γ_0 to γ_i
  - $\gamma_0 \rightarrow_1^* \gamma_i$ indicates that there is a **finite number of steps** in the execution from γ_0 to γ_i
  - $\gamma_0 \rightarrow_1^i \gamma_i$ and $\gamma_0 \rightarrow_1^* \gamma_i$ need **not** be derivation sequences

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Peter Müller—Semantics of Programming Languages, SS04 – p.109

Derivation Sequences: Example

- What is the final state if statement

  `z:=x; x:=y; y:=z`

  is executed in state $\{x \mapsto 5, y \mapsto 7, z \mapsto 0\}$?

$$\langle \texttt{z:=x; x:=y; y:=z}, \{x \mapsto 5, y \mapsto 7, z \mapsto 0\}\rangle$$
$$\rightarrow_1 \langle \texttt{x:=y; y:=z}, \{x \mapsto 5, y \mapsto 7, z \mapsto 5\}\rangle$$
$$\rightarrow_1 \langle \texttt{y:=z}, \{x \mapsto 7, y \mapsto 7, z \mapsto 5\}\rangle$$
$$\rightarrow_1 \{x \mapsto 7, y \mapsto 5, z \mapsto 5\}$$

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Derivation Trees

- Derivation trees explain why transitions take place
- For the first step

$$\langle \texttt{z:=x; x:=y; y:=z}, \sigma\rangle \rightarrow_1 \langle \texttt{x:=y; y:=z}, \sigma[\texttt{z} \mapsto 5]\rangle$$

the derivation tree is

$$\dfrac{\dfrac{\langle \texttt{z:=x}, \sigma\rangle \rightarrow_1 \sigma[\texttt{z} \mapsto 5]}{\langle \texttt{z:=x; x:=y}, \sigma\rangle \rightarrow_1 \langle \texttt{x:=y}, \sigma[\texttt{z} \mapsto 5]\rangle}}{\langle \texttt{z:=x; x:=y; y:=z}, \sigma\rangle \rightarrow_1 \langle \texttt{x:=y; y:=z}, \sigma[\texttt{z} \mapsto 5]\rangle}$$

- `z:=x; ( x:=y; y:=z )` would lead to a simpler tree with only one rule application

ETH Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Peter Müller—Semantics of Programming Languages, SS04 – p.111

Derivation Sequences and Trees

- Natural (big-step) semantics
  - The execution of a statement (sequence) is described by one big transition
  - The big transition can be seen as trivial derivation sequence with exactly one transition
  - The derivation tree explains why this transition takes place
- Structural operational (small-step) semantics
  - The execution of a statement (sequence) is described by one or more transitions
  - Derivation sequences are important
  - Derivation trees justify each individual step in a derivation sequence

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Termination

- The execution of a statement s in state σ
  - **terminates** iff there is a finite derivation sequence starting with $\langle s, \sigma \rangle$
  - **loops** iff there is an infinite derivation sequence starting with $\langle s, \sigma \rangle$

- The execution of a statement s in state σ
  - **terminates successfully** if $\langle s, \sigma \rangle \rightarrow_1^* \sigma'$
  - In IMP, an execution terminates successfully iff it terminates (no stuck configurations)

ETH Eidgenössische Technische Hochschule Zürich Swiss Federal Institute of Technology Zurich

Peter Müller—Semantics of Programming Languages, SS04 – p.113

Comparison: Summary

Natural Semantics	Structural Operational Semantics
▸ Local variable declarations and procedures can be modeled easily	▸ Local variable declarations and procedures require modeling the execution stack
▸ No distinction between abortion and looping	▸ Distinction between abortion and looping
▸ Non-determinism suppresses looping (if possible)	▸ Non-determinism does not suppress looping
▸ Parallelism cannot be modeled	▸ Parallelism can be modeled

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Peter Müller—Semantics of Programming Languages, SS04 – p.134

Motivation

- Operational semantics is at a rather low abstraction level
  - Some arbitrariness in choice of rules (e.g., size of steps)
  - Syntax involved in description of behavior
- Semantic equivalence in natural semantics

$$\langle s_1, \sigma \rangle \rightarrow \sigma' \Leftrightarrow \langle s_2, \sigma \rangle \rightarrow \sigma'$$

- Idea
  - We can describe the behavior on an abstract level if we are only interested in equivalence
  - We specify only the partial function on states

ETH Eidgenössische Technische Hochschule Zürich Swiss Federal Institute of Technology Zurich

Approach

- Denotational semantics describes the **effect** of a computation
- A semantic function is defined for each syntactic construct
  - maps syntactic construct to a mathematical object, often a function
  - the mathematical object describes the effect of executing the syntactic construct

ETH Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Compositionality

- In denotational semantics, semantic functions are defined **compositionally**
- There is a semantic clause for each of the basis elements of the syntactic category
- For each method of constructing a composite element (in the syntactic category) there is a semantic clause defined in terms of the **semantic function applied to the immediate constituents** of the composite element

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Peter Müller—Semantics of Programming Languages, SS04 – p.196

Examples

- The semantic functions $\mathcal{A}$: Aexp $\longrightarrow$ State $\longrightarrow$ Val and $\mathcal{B}$: Bexp $\longrightarrow$ State $\longrightarrow$ Bool are denotational definitions

$$
\begin{array}{lll}
\mathcal{A}[\![x]\!]\sigma & = \sigma(x) & \\
\mathcal{A}[\![i]\!]\sigma & = i & \text{for } i \in \mathbb{Z} \\
\mathcal{A}[\![e_1 \; op \; e_2]\!]\sigma & = \mathcal{A}[\![e_1]\!]\sigma \; \overline{op} \; \mathcal{A}[\![e_2]\!]\sigma & \text{for } op \in \text{ Op}
\end{array}
$$

$$
\mathcal{B}[\![e_1 \; op \; e_2]\!]\sigma = \begin{cases} tt & \text{if } \mathcal{A}[\![e_1]\!]\sigma \; \overline{op} \; \mathcal{A}[\![e_2]\!]\sigma \\ ff & \text{otherwise} \end{cases}
$$

Counterexamples

- The semantic functions $\mathcal{S}_{NS}$ and $\mathcal{S}_{SOS}$ are not denotational definitions because they are not defined compositionally

$$\mathcal{S}_{NS} : \text{Stm} \to (\text{State} \hookrightarrow \text{State})$$

$$\mathcal{S}_{NS}[\![s]\!]\sigma = \begin{cases} \sigma' & \text{if } \langle s, \sigma \rangle \to \sigma' \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$\mathcal{S}_{SOS} : \text{Stm} \to (\text{State} \hookrightarrow \text{State})$$

$$\mathcal{S}_{SOS}[\![s]\!]\sigma = \begin{cases} \sigma' & \text{if } \langle s, \sigma \rangle \to_1^* \sigma' \\ \text{undefined} & \text{otherwise} \end{cases}$$

ETH Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Peter Müller—Semantics of Programming Languages, SS04 – p.198

Semantic Functions

- The effect of executing a statement is described by the partial function $\mathcal{S}_{DS}$

$$\mathcal{S}_{DS} : \mathsf{Stm} \longrightarrow (\mathsf{State} \hookrightarrow \mathsf{State})$$

- Partiality is needed to model non-termination
- The effects of evaluating expressions is defined by the functions $\mathcal{A}$ and $\mathcal{B}$

ETH Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Peter Müller—Semantics of Programming Languages, SS04 – p.200

Direct Style Semantics of IMP

- `skip` does not modify the state

$$\mathcal{S}_{DS}[\![\texttt{skip}]\!] = id$$

$$id : \text{State} \longrightarrow \text{State}$$
$$id(\sigma) = \sigma$$

- x `:=` e assigns the value of e to variable x

$$\mathcal{S}_{DS}[\![x \texttt{:=} e]\!]\sigma = \sigma[x \mapsto \mathcal{A}[\![e]\!]\sigma]$$

ETH Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Peter Müller—Semantics of Programming Languages, SS04 – p.201

Direct Style Semantics of IMP (cont'd)

- Sequential composition $s_1\,;s_2$

$$\mathcal{S}_{DS}[\![s_1\,;s_2]\!] = \mathcal{S}_{DS}[\![s_2]\!] \circ \mathcal{S}_{DS}[\![s_1]\!]$$

- Function composition $\circ$ is defined in a **strict** way
  - If one of the functions is undefined on the given argument then the composition is undefined

$$(f \circ g)\sigma = \begin{cases} f(g(\sigma)) & \text{if } g(\sigma) \neq \text{undefined} \\ & \text{and } f(g(\sigma)) \neq \text{undefined} \\ \text{undefined} & \text{otherwise} \end{cases}$$

ETH Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Peter Müller—Semantics of Programming Languages, SS04 – p.202

Direct Style Semantics of IMP (cont'd)

- Conditional statement `if` b `then` s_1 `else` s_2 `end`

$$\mathcal{S}_{DS}[\![\texttt{if}\ b\ \texttt{then}\ s_1\ \texttt{else}\ s_2\ \texttt{end}]\!] = cond(\mathcal{B}[\![b]\!], \mathcal{S}_{DS}[\![s_1]\!], \mathcal{S}_{DS}[\![s_2]\!])$$

- The function $cond$
  - takes the semantic functions for the condition and the two statements
  - when supplied with a state selects the second or third argument depending on the first

$$cond : (\text{State} \rightarrow \text{Bool}) \times (\text{State} \hookrightarrow \text{State}) \times (\text{State} \hookrightarrow \text{State}) \rightarrow (\text{State} \hookrightarrow \text{State})$$

ETH Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Peter Müller—Semantics of Programming Languages, SS04 – p.203

Definition of $cond$

$$cond : (\text{State} \rightarrow \text{Bool}) \times (\text{State} \hookrightarrow \text{State}) \times (\text{State} \hookrightarrow \text{State}) \rightarrow (\text{State} \hookrightarrow \text{State})$$

$$cond(b, f, g)\sigma = \begin{cases} f(\sigma) & \text{if } b(\sigma) = tt \\ & \text{and } f(\sigma) \neq \text{undefined} \\ g(\sigma) & \text{if } b(\sigma) = ff \\ & \text{and } g(\sigma) \neq \text{undefined} \\ \text{undefined} & \text{otherwise} \end{cases}$$

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Peter Müller—Semantics of Programming Languages, SS04 – p.204

Semantics of Loop: Observations

- Defining the semantics of `while` is difficult
- The semantics of `while` b `do` s `end` must be equal to `if` b `then` s`;while` b `do` s `end else skip end`
- This requirement yields:

$$\mathcal{S}_{DS}[\![\texttt{while}\ b\ \texttt{do}\ s\ \texttt{end}]\!] = cond(\mathcal{B}[\![b]\!], \mathcal{S}_{DS}[\![\texttt{while}\ b\ \texttt{do}\ s\ \texttt{end}]\!] \circ \mathcal{S}_{DS}[\![s]\!], id)$$

- We cannot use this equation as a definition because it is not compositional

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Functionals and Fixed Points

$$\mathcal{S}_{DS}[\![\texttt{while}\ b\ \texttt{do}\ s\ \texttt{end}]\!] = cond(\mathcal{B}[\![b]\!], \mathcal{S}_{DS}[\![\texttt{while}\ b\ \texttt{do}\ s\ \texttt{end}]\!] \circ \mathcal{S}_{DS}[\![s]\!], id)$$

- The above equation has the form $g = F(g)$
  - $g = \mathcal{S}_{DS}[\![\texttt{while}\ b\ \texttt{do}\ s\ \texttt{end}]\!]$
  - $F(g) = cond(\mathcal{B}[\![b]\!], g \circ \mathcal{S}_{DS}[\![s]\!], id)$
- F is a **functional** (a function from functions to functions)
- $\mathcal{S}_{DS}[\![\texttt{while}\ b\ \texttt{do}\ s\ \texttt{end}]\!]$ is a **fixed point** of the functional F

Direct Style Semantics of IMP: Loops

- Loop statement `while` b `do` s `end`

$$\mathcal{S}_{DS}[\![\texttt{while}\ b\ \texttt{do}\ s\ \texttt{end}]\!] = \mathit{FIX}\ F$$
$$\text{where } F(g) = \mathit{cond}(\mathcal{B}[\![b]\!], g \circ \mathcal{S}_{DS}[\![s]\!], \mathit{id})$$

- We write $\mathit{FIX}\ F$ to denote the fixed point of the functional F:

$$\mathit{FIX} \ : \ ((\text{State} \hookrightarrow \text{State}) \rightarrow (\text{State} \hookrightarrow \text{State})) \rightarrow (\text{State} \hookrightarrow \text{State})$$

- This defintion of $\mathcal{S}_{DS}[\![\texttt{while}\ b\ \texttt{do}\ s\ \texttt{end}]\!]$ is compositional

Example

- Consider the statement

```
while x # 0 do skip end
```

- The functional for this loop is defined by

$$\begin{aligned} F'(g)\sigma &= cond(\mathcal{B}[\![\texttt{x\#0}]\!], g \circ \mathcal{S}_{DS}[\![\texttt{skip}]\!], id)\sigma \\ &= cond(\mathcal{B}[\![\texttt{x\#0}]\!], g \circ id, id)\sigma \\ &= cond(\mathcal{B}[\![\texttt{x\#0}]\!], g, id)\sigma \\ &= \begin{cases} g(\sigma) & \text{if } \sigma(x) \neq 0 \\ \sigma & \text{if } \sigma(x) = 0 \end{cases} \end{aligned}$$

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Example (cont'd)

- The function

$$g_1(\sigma) = \begin{cases} \text{undefined} & \text{if } \sigma(x) \neq 0 \\ \sigma & \text{if } \sigma(x) = 0 \end{cases}$$

is a fixed point of F'

- The function $g_2(\sigma) = \text{undefined}$ is not a fixed point for F'

ETH Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Well-Definedness

$$\mathcal{S}_{DS}[\![\texttt{while}\ b\ \texttt{do}\ s\ \texttt{end}]\!] = FIX\ F$$
$$\text{where } F(g) = cond(\mathcal{B}[\![b]\!], g \circ \mathcal{S}_{DS}[\![s]\!], id)$$

- The function $\mathcal{S}_{DS}[\![\texttt{while}\ b\ \texttt{do}\ s\ \texttt{end}]\!]$ is well-defined if $FIX F$ defines a **unique fixed point** for the functional F
  - There are functionals that have more than one fixed point
  - There are functionals that have no fixed point at all

ETH Eidgenössische Technische Hochschule Zürich Swiss Federal Institute of Technology Zurich

Peter Müller—Semantics of Programming Languages, SS04 – p.211

Examples

- F' from the previous example has more than one fixed point

$$F'(g)\sigma = \begin{cases} g(\sigma) & \text{if } \sigma(x) \neq 0 \\ \sigma & \text{otherwise} \end{cases}$$

  - Every function g' : State $\hookrightarrow$ State with $g'(\sigma) = \sigma$ if $\sigma(x) = 0$ is a fixed point for F'

- The functional F_1 has no fixed point if $g_1 \neq g_2$

$$F_1(g) = \begin{cases} g_1 & \text{if } g = g_2 \\ g_2 & \text{otherwise} \end{cases}$$

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Peter Müller—Semantics of Programming Languages, SS04 – p.212

Hoare Logic

Hoare axioms and rules for simple while languages

- { P } **skip** { P }
- { P[x/e] } **x := e** { P }
- { P } **c1** { R } , { R } c2 { Q } ==> { P } **c1;c2** { Q }
- { P ∧ b } **c1** { Q } , { P ∧ !b } **c2** { Q } ==>
  { P } **if b then c1 else c2** { Q }
- { INV ∧ b } **c** { INV } ==> { INV } **while b do c** { INV ∧ !b }
- P –> P' , { P' } **c** { Q' } , Q' –> Q ==> { P } **c** { Q }
- **Semantics of the Hoare Logic**:
- { P } **c** { Q } == (ALL s. (P(s) ∧ s -**c**-> t) –> P(t))

Hoare Logic

Example

```
{ 0 <= x }
  c  := 0 ;
  sq := 1 ;
  WHILE sq <= x DO (*INV=(c*c <= x&sq=(c+1)*(c+1))*)
     c := c + 1 ;
     sq :=  sq + (2*c + 1);
{ c*c <= x  &  x < (c+1)*(c+1) }
```

Demo: MyHoare.thy

Chapter 9

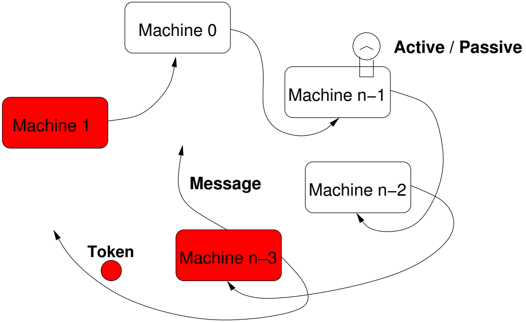
Application: Verification of distributed systems

Distributed Termination Detection : Dijkstra

Example 9.1. Implement the following termination detection protocol:

A passive machine becomes active, iff it receives a message from another machine.

Only active machines can send messages.



Edsger W. Dijkstra, W. H. J. Feijen, and A.J.M. van Gasteren. Derivation of a Termination Detection Algorithm for Distributed Computations. IPL 16 (1983).

Assumptions for distributed termination detection

Rules for a probe

Rule 0 When active, $Machine_{i+1}$ keeps the token; when passive, it hands over the token to $Machine_i$.

Rule 1 A machine sending a message makes itself red.

Rule 2 When $Machine_{i+1}$ propagates the probe, it hands over a red token to $Machine_i$ when it is red itself, whereas while being white it leaves the color of the token unchanged.

Rule 3 After the completion of an unsuccessful probe, $Machine_0$ initiates a next probe.

Rule 4 $Machine_0$ initiates a probe by making itself white and sending to $Machine_{n-1}$ a white token.

Rule 5 Upon transmission of the token to $Machine_i$, $Machine_{i+1}$becomes white. (Notice that the original color of $Machine_{i+1}$ may have affected the color of the token).

Correctness of the abstract version: Dijkstra

Assumptions

The machines constitute a closed system, i.e. messages can only be dispatched among each other (no outside messages). The system in the initial state can have any color and several machines can be active. The token is located in the 0'th. machine.

The given rules describe the transfer of the token and the coloration of the machines upon certain activities.

The task is to determine a state in which all the machines are passive (not active). This is a stable state of the system, because only active machines can dispatch messages and passive machines can only become active by receiving a message.

The invariant: Let t be the position on which the token is, then following invariant holds:

$(\forall i : t < i < n \;\; Machine_i \text{ is passive}) \vee (\exists j : 0 \leq j \leq t \;\; Machine_j \text{ is red}) \vee$
$(Token \text{ is red})$

Distributed Termination Detection: Correctness

$(\forall i : t < i < n \;\; Machine_i \text{ is passive}) \vee (\exists j : 0 \leq j \leq t \;\; Machine_j \text{ is red}) \vee$
$(Token \text{ is red})$

Correctness argument
When the token reaches $Machine_o$, $t = 0$ and the invariant holds.
If
$(Machine_o \text{ is passive}) \wedge (Machine_o \text{ is white}) \wedge (Token \text{ is white})$
then
$(\forall i : 0 < i < n \;\; Machine_i \text{ is passive})$ must hold, i.e. termination.

Proof of the invariant Induction over t:
The case t = n - 1 is easy.
Assume the invariant is valid for $0 < t < n$, prove it is valid for $t - 1$.

Distributed Abstract State Machines: Model

Signature:

static

$COLOR = \{red, white\}$ $TOKEN = \{redToken, whiteToken\}$
$MACHINE = \{0, 1, 2, \ldots, n-1\}$
$next : MACHINE \to MACHINE$
e.g. with $next(0) = n-1, next(n-1) = n-2, \ldots, next(1) = 0$

controlled

$color : MACHINE \to COLOR$ $token : MACHINE \to TOKEN$
$RedTokenEvent, WhiteTokenEvent : MACHINE \to BOOL$

monitored $Active : MACHINE \to BOOL$
$SendMessageEvent : MACHINE \to BOOL$

Distributed Termination Detection: DASM-Procedure

Macros: (Rule definitions)

- $ReactOnEvents(m : MACHINE) =$
  $if\ \ RedTokenEvent(m)\ \ then$
  $\quad token(m) := redToken$
  $\quad RedTokenEvent(m) :=\ undef$
  $if\ \ WhiteTokenEvent(m)\ \ then$
  $\quad token(m) := whiteToken$
  $\quad WhiteTokenEvent(m) :=\ undef$
  $if\ \ SendMessageEvent(m)\ \ then\ \ color(m) := red$ Rule 1

- $Forward(m : MACHINE, t : TOKEN) =$
  $if\ \ t = whiteToken\ \ then$
  $\quad WhiteTokenEvent(next(m)) :=\ true$
  $else$
  $\quad RedTokenEvent(next(m)) := true$

Distributed Termination Detection: DASM-Procedure

Programs

- $RegularMachineProgram =$

  $ReactOnEvents(me)$
  $if \neg\ Active(me) \wedge\ token(me) \neq\ undef\ \ then$ Rule 0
  $\quad InitializeMachine(me)$ Rule 5
  $\quad if\ \ color(me) =\ red\ \ then$
  $\quad\quad Forward(me, redToken)$ Rule 2
  $\quad else$
  $\quad\quad Forward(me, token(me))$ Rule 2

- With $InitializeMachine(m : MACHINE) =$

  $token(m) := undef$
  $color(m) := white$

Distributed Termination Detection: Procedure

Programs

- $SupervisorMachineProgram =$

$$
\begin{array}{l}
ReactOnEvents(me) \\
\textit{if} \neg\ Active(me) \wedge\ token(me) \neq\ undef\ \ then \\
\quad \textit{if}\ \ color(me) =\ white \wedge\ token(me) =\ whiteToken\ \ then \\
\quad\quad ReportGlobalTermination \\
\quad else \quad \text{Rule 3} \\
\quad\quad InitializeMachine(me) \quad \text{Rule 4} \\
\quad\quad Forward(me, whiteToken) \quad \text{Rule 4}
\end{array}
$$

Distributed Termination Detection

Initial states

$$\exists m_0 \in MACHINE$$
$$(program(m_0) = SupervisorMachineProgram \wedge$$
$$token(m_0) = redToken \wedge$$
$$(\forall m \in MACHINE)(m \neq m_0 \Rightarrow$$

$$(program(m) = RegularMachineProgram \wedge token(m) = undef)))$$

Environment constraints For all the executions and all linearizations holds:

$$\mathbf{G}\ (\forall m \in MACHINE)$$
$$(SendMessageEvent(m) = true \Rightarrow (\mathbf{P}(Active(m)) \ \wedge Active(m)))$$
$$\wedge\ ((Active(m) = true \wedge \mathbf{P}(\neg Active(m)) \Rightarrow$$
$$(\exists m' \in MACHINE)\ (m' \neq m \wedge\ SendMessageEvent(m'))))$$

Nextconstraints

Chapter 10

Conclusions: Overall structure

Overall structure

1. Introduction
2. Functional specification and programming
3. Language and semantical aspects of higher-order logic
4. Proof system for higher-order logic
5. Sets, functions, relations, and fixpoints
6. Verifying functions
7. Inductively defined sets
8. Specification of programming language semantics
9. Program verification and programming logic

Chapter 1: Introduction

1. Give an overview of the course.
2. Explain the terms model, specification, verification.
3. Explain language and semantics of propositional logic.
4. Give and explain a logical rule. How is this rule applied?
5. What is a Hilbert style, what a natural deduction style proof system?
6. What is the advantage of a Hilbert style proof system?
7. Why is a natural deduction style proof system chosen for interactive proof assistants?

Chapter 2: Functional programming and specification

1. What is the relationship between the data type construct and the case expression? Illustrate the relationship by an example.
2. What is the meaning of “fun f x = f x” in ML, what is the meaning of the corresponding definition in Isabelle/HOL?
3. Why are there different forms of function definitions in Isabelle/HOL, but only one in ML?
4. Why is there a distinction between types with equality and types without equality in ML, but not in Isabelle/HOL?

Chapter 3: Language and semantical aspects of HOL

1. What is the foundational reason that HOL is typed? Are there other reasons w.r.t. an application in computer science?
2. What does “higher-order” mean?
3. Why is predicate logic not sufficient? Give an example?
4. What are the types in HOL?
5. What are the terms in HOL? Give examples of constants.
6. Explain the description operator.
7. What is a frame? What is an interpretation?
8. How is satisfiability defined?

9. What is a standard model?
10. Give and explain one of the axioms of HOL?
11. Can the constants True and False be defined in HOL?
12. What does it mean that HOL+infinity is incomplete wrt. standard models?
13. What is a conservative extension?
14. What is the advantage of conservative extensions over axiomatic definitions?
15. Which syntactic schemata for conservative extensions were treated in the lecture?
16. Give examples of constant definitions.
17. Explain the definitions of new types?
18. Does a data type definition in Isabelle/HOL lead to a new type?

Chapter 4: Proof system for HOL

1. A natural deduction proof system distinguishes between formulas, sequents, and rules. What are the differences?
2. Isabelle/HOL has nor clear distinction between sequents and rules. Why?
3. Explain the different kinds of variables.
4. What is a proof state?
5. What is the distinction between a rule and a method?
6. Explain the method “rule” and show in detail how it can be applied in a proof state?
7. What is an elimination rule?
8. Here is a proof state (shown on the screen). What is a rule that can be applied?

9. Name some rule and their uses.
10. What does it mean that a rule is safe?
11. Why is verification in Isabelle/HOL usually based on theory Main and not directly on the HOL axioms?
12. What is rewriting and simplification?
13. How can an Isabelle/HOL user influence the simplification process?
14. What is case analysis?
15. How differ methods for proof automation?
16. Explain a method for proof automation.
17. What is a forward proof step?

Chapter 5: Sets, functions, relations, and fixpoints

1. What is the relationship between sets and functions?
2. What is set comprehension?
3. How are sets be realized in Isabelle/HOL?
4. Whare is the relationship between sets and types (in Isabelle/HOL)?
5. What is the principle of extensionality for functions? Why is it important for verification?
6. Define injectivity as a predicate in Isabelle/HOL.
7. How are relations represented in Isabelle/HOL. What would be a different representation?

8. How can the reflexive and transitive closure of a relation be defined? Can this be done in first order logic?
9. What is a well-founded relation?
10. What is a measure function?
11. Explain an application of well-founded relations?
12. What is a complete lattice? Give an example of a complete lattice.
13. Explain the Kaster/Tarski theorem. Why is it important? What is the relationship to inductive definitions?

Chapter 6: Verifying functions

1. Explain the difference between verification and testing.
2. What is the advantage of formal proofs over paper and pencil proofs?
3. Property specifications can be wrong. Does this mean that verification is useless?
4. What is the relationship between termination and well-definedness of functions?
5. How is termination usually proved? Sketch this for gcd and quicksort.
6. What are the properties we proved for quicksort?

7. Explain shallow embedding.
8. How can functional properties of algorithms are proven in Isabelle/HOL?
9. Can Isabelle/HOL be used to prove the complexity of an algorithm? What would be needed (together with Chapter 8)?
10. What does structural induction over the function parameters mean?

Chapter 7: Inductively defined sets

1. Explain the inductive definition of sets. What is the syntactic schema used?
2. Why is it necessary to constrain inductive definition to the syntactic schema?
3. Give an example of an inductive definition.
4. What is the relationsship between recursive and inductive definitions?
5. What is a coinductive definition?

6. For which situation are coinductive definitions needed?
7. What is a transition system? Give examples.
8. Explain the syntax of LTL defined in the lecture.
9. What is a Kripke structure? How is it related to transition systems?
10. What is a liveness property?

Chapter 8:
Specification of programming language semantics

1. What is a programming language semantics? Who is a typical user of a semantics?
2. What is a deep embedding of a language into a specification framework such as Isabelle/HOL?
3. Explain big step semantics.
4. What can be expressed in small step semantics that is not directly expressable in big step semantics?

5. Show how the semantics of parallel statement execution can be handled in small step semantics.
6. What does compositionality mean in the context of denotational semantics?
7. How is operational semantics formalized in Isabelle/HOL? Explain motivations for such formalizations.
8. Can programming language semantics be used for program verification?

Chapter 9:
Program verification and programming logic

1. What does it mean that a Hoare triple is valid? How can validity be formalized?
2. How can a programming logic be expressed in HOL?
3. Why are assertions in Hoare logic be formalized as functions?
4. Can Hoare logic proofs be done in Isabelle/HOL? Explain a rule application?
5. What does soundness mean for a Hoare logic? How is soundness proved?