COMP4011/8011
Advanced Topics in
Formal Methods and Programming Languages

– Software Verification with Isabelle/HOL –

Peter Höfner

September 22, 2024

Section 0

Admin

Lecturer

- **A/Prof. Peter Höfner**
  CSIT, Room N234 (Building 108)
  Peter.Hoefner@anu.edu.au
  +61 2 6125 0159

  Consultation
  after the lecture, or by appointment

Lectures

- Tuesday 10:00-11:30, Rm G51 Haydon-Allen Bldg
  Wednesday 9:00-10:30, Rm G52 Haydon-Allen Bldg
- Q/A session in Week 12 or 13

- **Etiquette**
  - tailored for in-person attendance
  - engage
  - feel free to ask questions
  - we reject behaviour that strays into harassment, no matter how mild

DropIns

- not mandatory
- Thursday 11:00-13:30, Rm G51 Haydon-Allen Bldg
- from Week 2 onwards

- **Summary**
  - your chance to discuss problems
  - discuss home work
  - discuss exercises from lectures
  - **no structured activity**
    (nothing will happen without your input)
  - except: oral discussion of your assignments

Plan/Schedule I

Resources

web: `https://cs.anu.edu.au/courses/comp4011-itp/`
wattle: `https://wattlecourses.anu.edu.au/course/view.php?id=44081`
edstem: `https://edstem.org/`
(you will be registered at the end of the week)

Workload

The average student workload is 130 hours for a six unit course.
That is roughly **11 hours/week**.
`https://policies.anu.edu.au/ppl/document/ANUP_000691`

Plan/Schedule II

Assessment criteria

- Quizz: 0% (for feedback only)
- Assignments: 45%, 3 assignments
- Take-home exam: 55% (55 marks) **[hurdle]**
- **hurdle:** minimum of 35% in the final exam

Assessments (tentative)

No	Hand Out	Hand In	Marks
0	23/07	26/07	0
1		16/08	15
2		04/10	15
3		12/10	15
4		tbc	55

About the Course I

This course is about mechanical proof assistants, how they work, and what they can be used for. It presents specification and proof techniques used in industrial grade interactive theorem provers (Isabelle/HOL), teaches the theoretical background to the techniques involved, and shows how to use a theorem prover to conduct formal proofs in practice.

About the Course II

Topics (tentative)

The following schedule is tentative and likely to change.

	Topic
0	Admin
1	Introduction
2	Lambda Calculus
3	Proofs in Isabelle, Natural Deduction, HOL
4	Term Rewriting
5	Induction
6	Recursive Datatypes and Primitive Recursion
7	General Recursion
8	Hoare Logic
9	Weakest Preconditions
10	Advanced Topics in Software Verification
11	Guest lectures and Exam Preparation

About the Course IV

Disclaimer

This is first time I offer this course.

The material in these notes has been drawn from several different sources, including the books and similar courses at some other universities. In particular, it is based on a course offered by UNSW and Proofcraft.

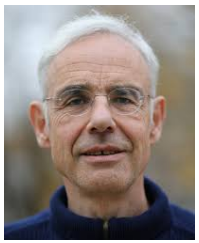
As it is a newly designed course, changes in timetabling are quite likely.

Feedback (oral, email, survey, . . .) is highly appreciated.

Credits



Gerwin Klein, June Andronick, Johannes Åman Pohjola



Tobias Nipkow, Larry Paulson, Makarius Wenzel


David Basin, Burkhardt Wolff

Academic Integrity

- never misrepresent the work of others as your own
- if you take ideas from elsewhere (including tools) you must say so with utmost clarity

Generative AI

- introduction of fundamental concepts
- use of any Generative AI tools is not permitted

Reading Material

- Tobias Nipkow and Gerwin Klein: Concrete Semantics
  `http://www.concrete-semantics.org`

Further Reading

- Tobias Nipkow, Lawrence C. Paulson, Markus Wenzel: Isabelle/HOL – A Proof Assistant for Higher-Order Logic
- Apostolos Doxiadis, Christos H. Papadimitriou, Alecos Papadatos, Annie Di Donna. Logicomix: An Epic Search for Truth
- Hendrik Pieter Barendregt. The Lambda Calculus, its Syntax and Semantics
- Alonzo Church. A formulation of the simple theory of types
- Michael J. C. Gordon and Tom F. Melham (eds), Introduction to HOL. Cambridge University Press
- Franz Baader and Tobias Nipkow. Term Rewriting and All That
- ...

Software

- Isabelle (Australian download mirror)
  `https://proofcraft.systems/isabelle/index.html`
- Isabelle theory library
  `https://isabelle.in.tum.de/library/`
- The Archive of Formal Proofs
  `https://www.isa-afp.org`

Exercise 1

Install Isabelle
Feel free to bring your laptop into lectures and dropins.

Section 1

Introduction

Binary Search (`java.util.Arrays`)

```
1:     public static int binarySearch(int[] a, int key) {
2:         int low = 0;
3:         int high = a.length - 1;
4:
5:         while (low <= high) {
6:             int mid = (low + high) / 2;
7:             int midVal = a[mid];
8:
9:             if (midVal < key)
10:                 low = mid + 1
11:             else if (midVal > key)
12:                 high = mid - 1;
13:             else
14:                 return mid; // key found
15:         }
16:         return -(low + 1);  // key not found.
17:     }
```

6: `int mid = (low + high) / 2;`

http://googleresearch.blogspot.com/2006/06/extra-extra-read-all-about-it-nearly.html

What you will learn

- how to use a theorem prover
- background, how it works
- how to prove and specify
- how to reason about programs

Health Warning

Theorem Proving may be addictive

Prerequisites

This is an advanced course. It assumes knowledge in

- Functional programming
- First-order formal logic

The following program should make sense to you:

```
map f []     =  []
map f (x:xs) =  f x : map f xs
```

You should be able to read and understand this formula:

$$\exists x.\ (P(x) \longrightarrow \forall x.\ P(x))$$

Increase chance to succeed

you should:

- attend lectures
- try Isabelle early
- redo all the demos alone
- try the exercises/homework we give, when we do give some
- **DO NOT CHEAT**
  - assignments and exams are take-home. This does NOT mean you can work in groups. Each submission is personal.
  - for more info, see Plagiarism Policy

What is a formal proof?

A derivation in a formal calculus

Example: $A \wedge B \longrightarrow B \wedge A$ derivable in the following system

Rules:

$$\frac{X \in S}{S \vdash X}\ \text{(assumption)} \qquad \frac{S \cup \{X\} \vdash Y}{S \vdash X \longrightarrow Y}\ \text{(impI)}$$

$$\frac{S \vdash X \quad S \vdash Y}{S \vdash X \wedge Y}\ \text{(conjI)} \qquad \frac{S \cup \{X, Y\} \vdash Z}{S \cup \{X \wedge Y\} \vdash Z}\ \text{(conjE)}$$

Proof:

1.	$\{A, B\} \vdash B$	(by assumption)
2.	$\{A, B\} \vdash A$	(by assumption)
3.	$\{A, B\} \vdash B \wedge A$	(by conjI with 1 and 2)
4.	$\{A \wedge B\} \vdash B \wedge A$	(by conjE with 3)
5.	$\{\} \vdash A \wedge B \longrightarrow B \wedge A$	(by impI with 4)

What is a theorem prover?

Implementation of a formal logic on a computer.

- fully automated (propositional logic)
- automated, but not necessarily terminating (first order logic)
- with automation, but mainly interactive (higher order logic)

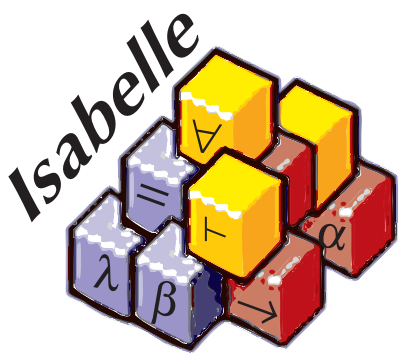
There are other (algorithmic) verification tools:

- model checking, static analysis, ...
- See COMP3710: Algorithmic Verification (S2 2022) or COMP4130

Why theorem proving?

- Analyse systems/programs thoroughly
- Find design and specification errors early
- High assurance: mathematical, machine checked proofs
- It's not always easy
- It's fun!

Main theorem proving system for this course



Isabelle

What is Isabelle?

A generic interactive proof assistant

- **generic**
  not specialised to one particular logic
  (two large developments: HOL and ZF, will mainly use HOL)
- **interactive**
  more than just yes/no, you can interactively guide the system
- **proof assistant**
  helps to explore, find, and maintain proofs

Correctness

If I prove it on the computer, it is correct, right?

No, because:

1. hardware could be faulty
2. operating system could be faulty
3. implementation runtime system could be faulty
4. compiler could be faulty
5. implementation could be
6. logic could be inconsistent
7. theorem could mean something else

Correctness

If I prove it on the computer, it is correct, right?

No, but: probability for

- OS and H/W issues reduced by using different systems
- runtime/compiler bugs reduced by using different compilers
- faulty implementation reduced by having the right prover architecture
- inconsistent logic reduced by implementing and analysing it
- wrong theorem reduced by expressive/intuitive logics

No guarantees, but assurance immensely higher than manual proof

Meta Logic

Meta language:
The language used to talk about another language.

Examples:
English in a Spanish class, English in an English class

Meta logic:
The logic used to formalise another logic

Example:
Mathematics used to formalise derivations in formal logic

Meta Logic – Example

Syntax:

Formulae: $F ::= V \mid F \longrightarrow F \mid F \wedge F \mid \mathit{False}$
$V ::= [A - Z]$

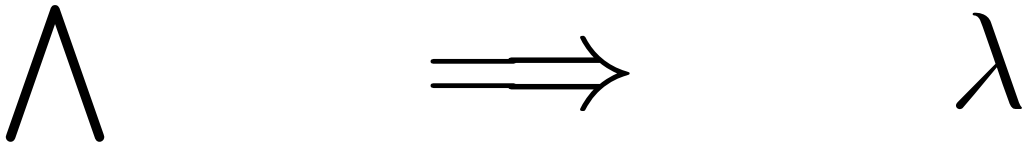
Judgement: $S \vdash X$ X a formula, S a set of formulae

logic / meta logic

$$\frac{X \in S}{S \vdash X} \qquad \frac{S \cup \{X\} \vdash Y}{S \vdash X \longrightarrow Y}$$

$$\frac{S \vdash X \quad S \vdash Y}{S \vdash X \wedge Y} \qquad \frac{S \cup \{X, Y\} \vdash Z}{S \cup \{X \wedge Y\} \vdash Z}$$

Isabelle's Meta Logic



$\bigwedge$

Syntax: $\bigwedge x.\ F$ (F another meta logic formula)
in ASCII: `!!x. F`

- this is the meta-logic universal quantifier
- example and more later

$\Longrightarrow$

Syntax: $A \Longrightarrow B$ (A, B other meta logic formulae)
in ASCII: `A ==> B`

Binds to the right:

$$A \Longrightarrow B \Longrightarrow C \quad = \quad A \Longrightarrow (B \Longrightarrow C)$$

Abbreviation:

$$[\![A; B]\!] \Longrightarrow C \quad = \quad A \Longrightarrow B \Longrightarrow C$$

- read: A and B implies C
- used to write down rules, theorems, and proof states

Example: a theorem

mathematics: if $x < 0$ and $y < 0$, then $x + y < 0$

formal logic: $\vdash\ x < 0 \wedge y < 0 \longrightarrow x + y < 0$
variation: $x < 0; y < 0\ \vdash\ x + y < 0$

Isabelle: **lemma** "$x < 0 \wedge y < 0 \longrightarrow x + y < 0$"
variation: **lemma** "$[\![x < 0; y < 0]\!] \Longrightarrow x + y < 0$"
variation: **lemma**
assumes "$x < 0$" and "$y < 0$" shows "$x + y < 0$"

Example: a rule

logic: $\dfrac{X \quad Y}{X \wedge Y}$

variation: $\dfrac{S \vdash X \quad S \vdash Y}{S \vdash X \wedge Y}$

Isabelle: $[\![X; Y]\!] \Longrightarrow X \wedge Y$

Example: a rule with nested implication

logic:

$$\dfrac{X \vee Y \quad \begin{array}{c}[X]\\ \vdots\\ Z\end{array} \quad \begin{array}{c}[Y]\\ \vdots\\ Z\end{array}}{Z}$$

variation:

$$\dfrac{S \cup \{X\} \vdash Z \quad S \cup \{Y\} \vdash Z}{S \cup \{X \vee Y\} \vdash Z}$$

Isabelle:

$$[\![X \vee Y; X \Longrightarrow Z; Y \Longrightarrow Z]\!] \Longrightarrow Z$$

λ

Syntax: $\lambda x.\ F$ (F another meta logic formula)
in ASCII: `%x. F`

- lambda abstraction
- used to represent functions
- used to encode bound variables
- more about this soon

Section 2

Enough Theory!

Getting started with Isabelle

System Architecture

Prover IDE (jEdit) – user interface

HOL, ZF – object-logics

Isabelle – generic, interactive theorem prover

Standard ML – logic implemented as ADT

User can access all layers!

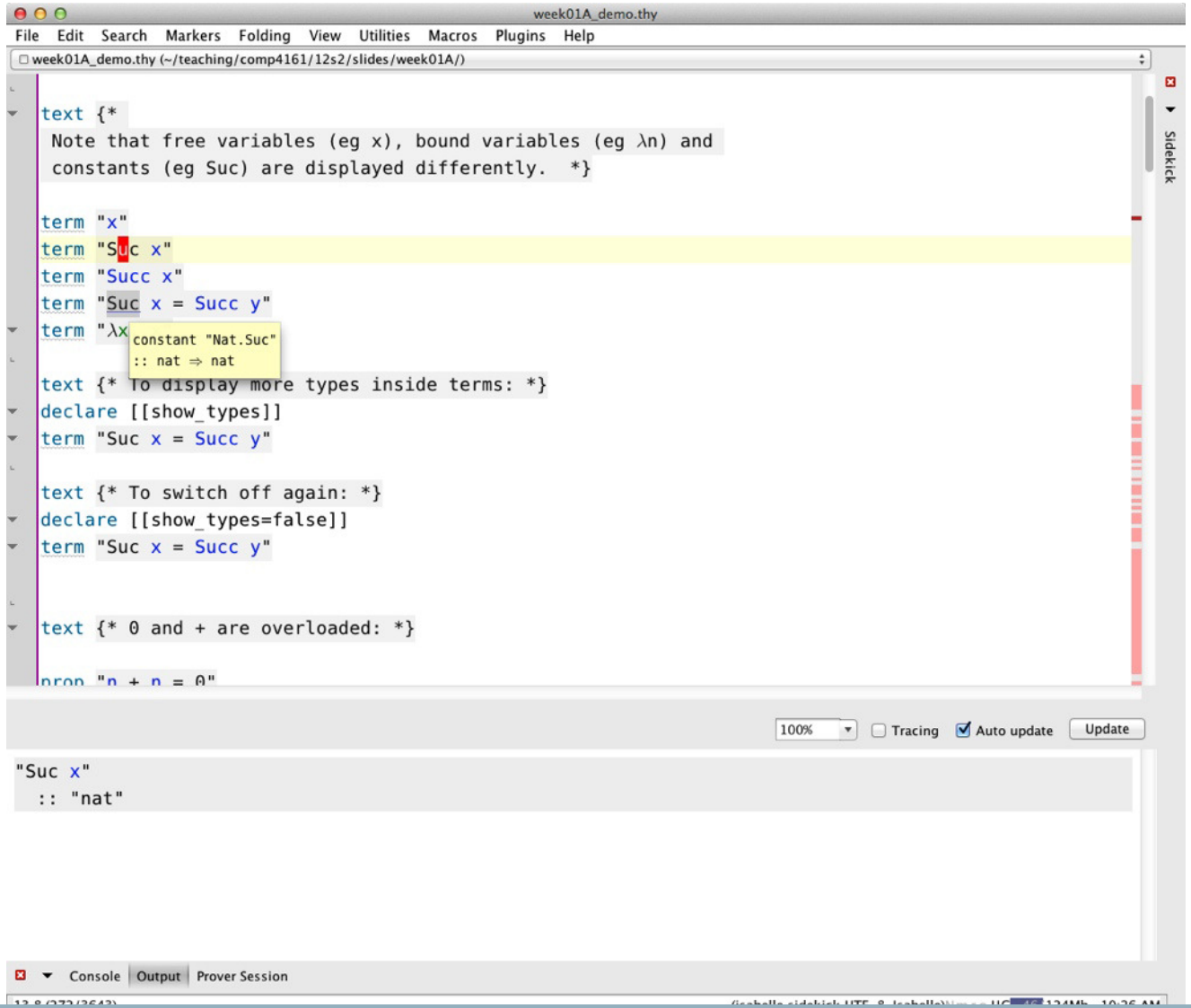
System Requirements

- Linux, Windows, or MacOS X (10.8 +)
- Standard ML (PolyML implementation)
- Java (for jEdit)

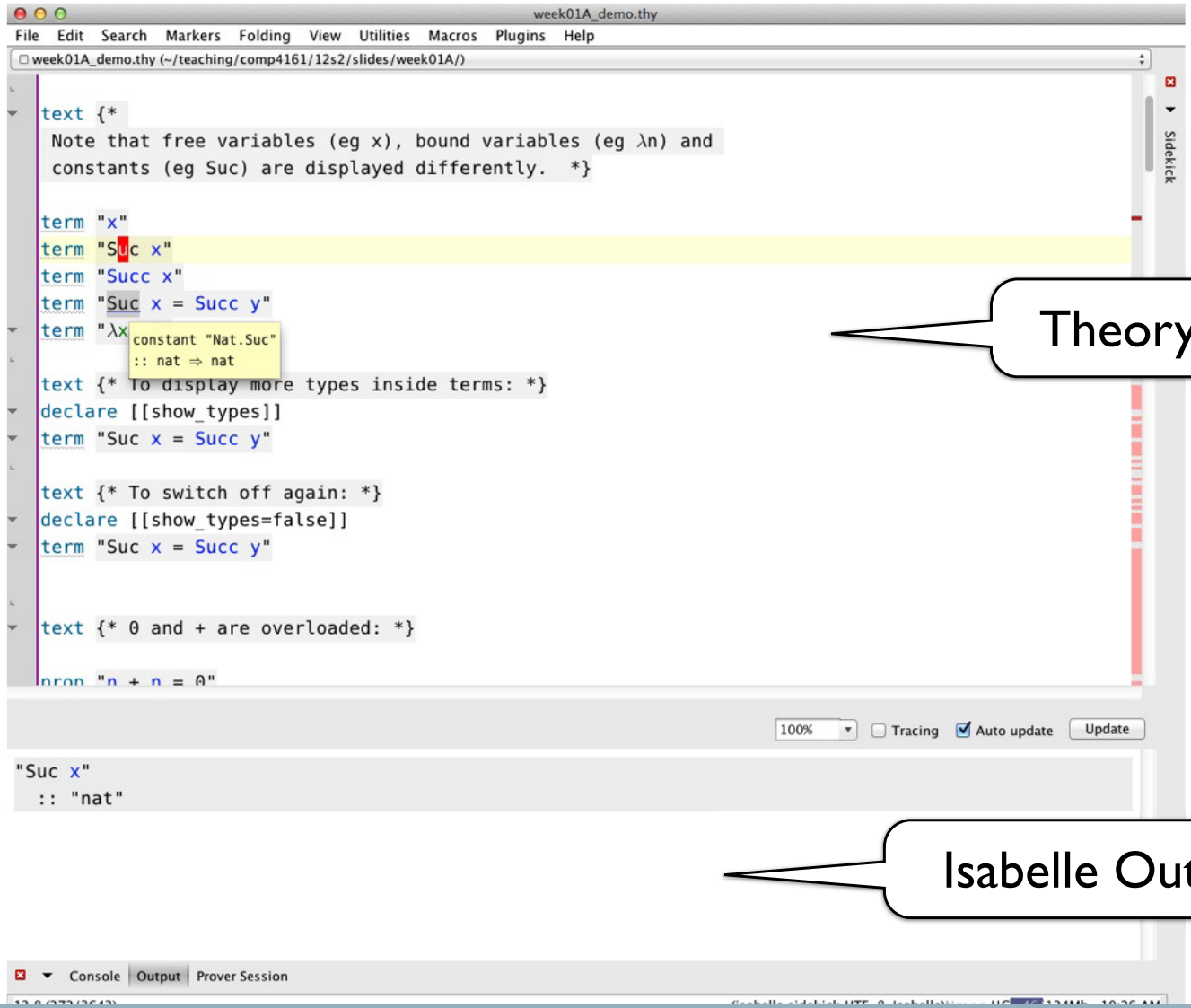
Pre-made packages for Linux, Mac, and Windows + info on:
`https://proofcraft.systems/isabelle/`

Demo

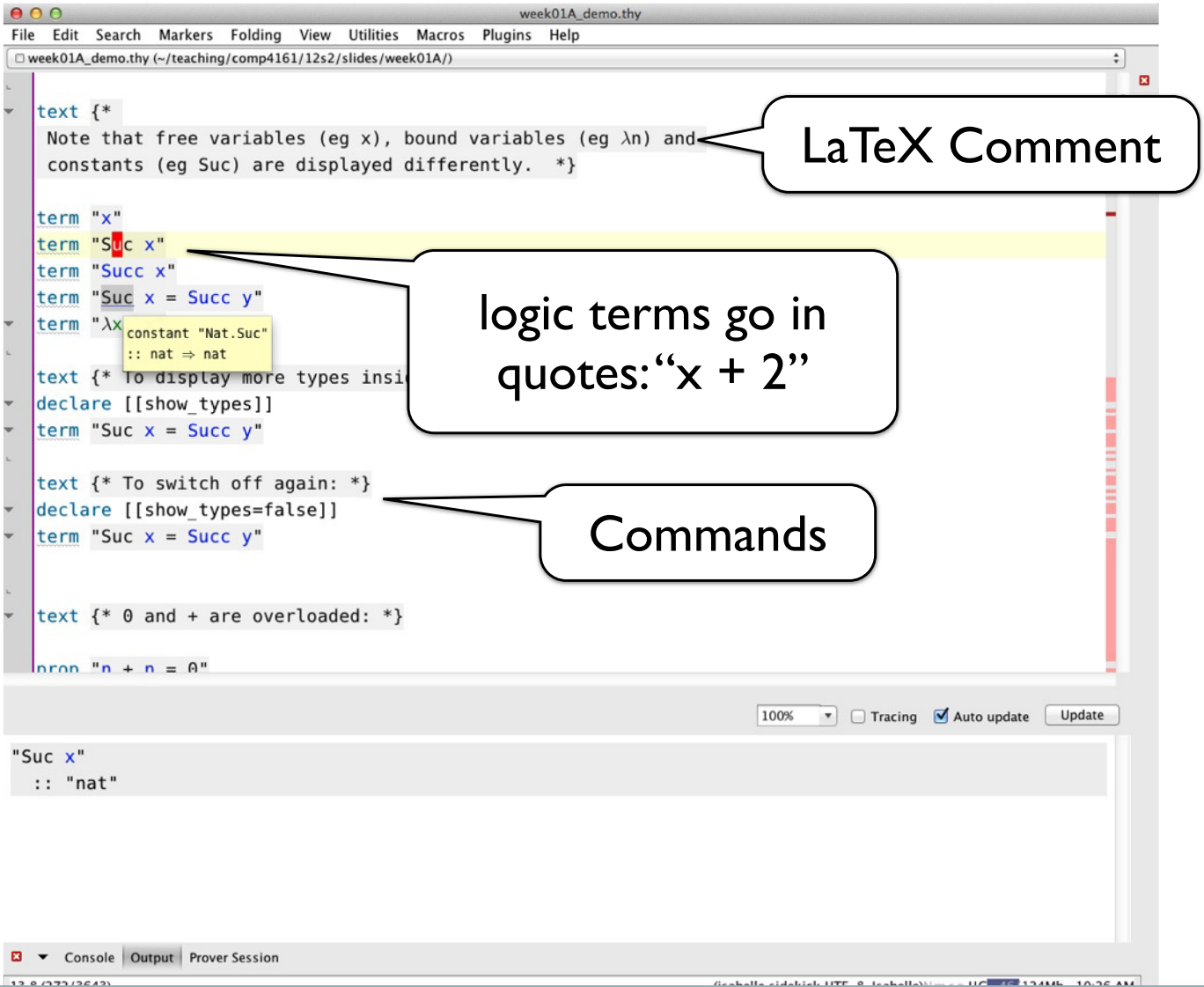
jEdit/PIDE


week01A_demo.thy
File Edit Search Markers Folding View Utilities Macros Plugins Help
week01A_demo.thy (~/teaching/comp4161/12s2/slides/week01A/)
text {*
Note that free variables (eg x), bound variables (eg λn) and
constants (eg Suc) are displayed differently. *}
term "x"
term "Suc x"
term "Succ x"
term "Suc x = Succ y"
term "λx
constant "Nat.Suc"
:: nat ⇒ nat
text {* To display more types inside terms: *}
declare [[show_types]]
term "Suc x = Succ y"
text {* To switch off again: *}
declare [[show_types=false]]
term "Suc x = Succ y"
text {* 0 and + are overloaded: *}
prop "n + n = 0"
Sidekick
100%
Tracing
Auto update
Update
"Suc x"
:: "nat"
Console
Output
Prover Session

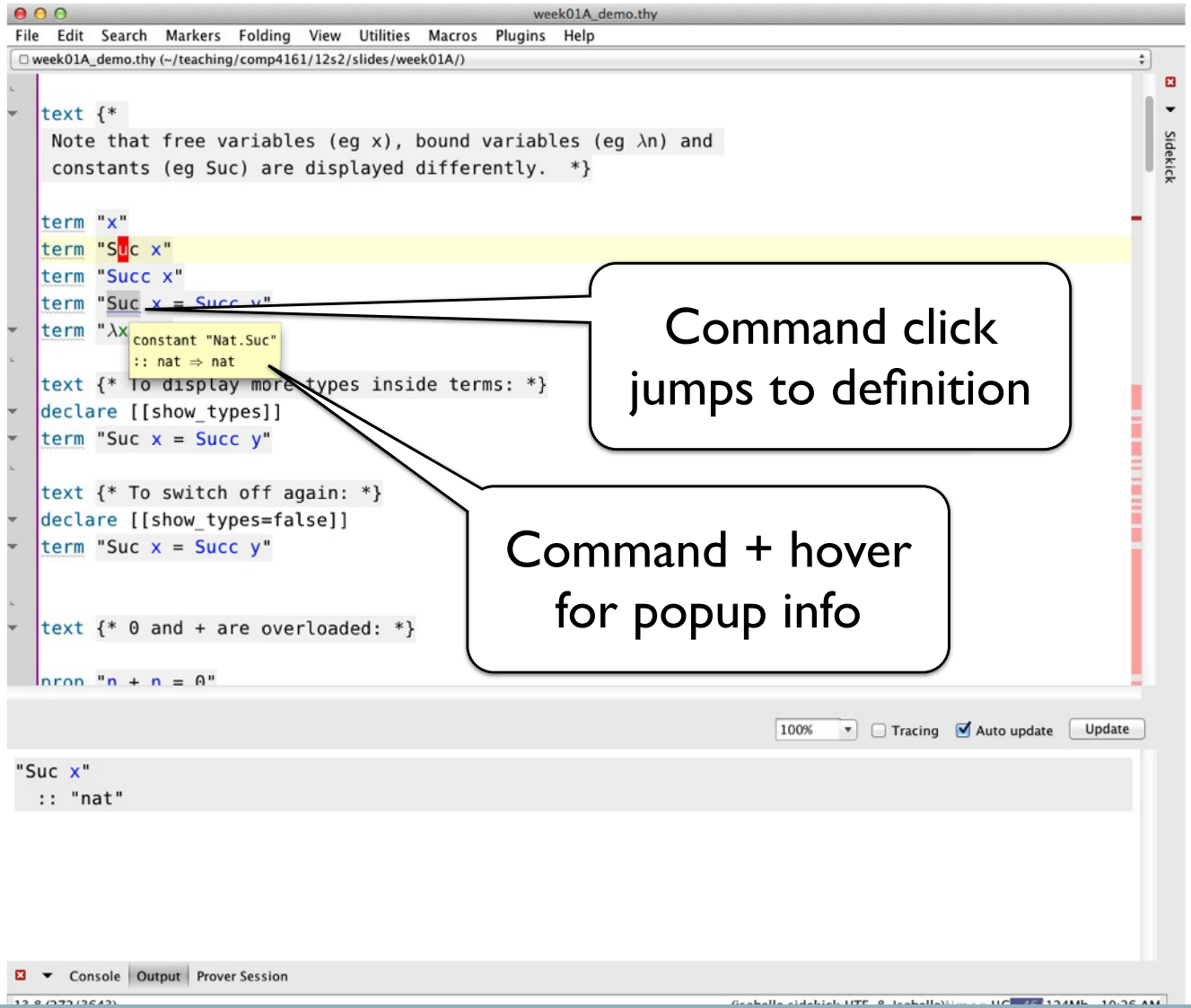
jEdit/PIDE


week01A_demo.thy
File Edit Search Markers Folding View Utilities Macros Plugins Help
week01A_demo.thy (~/teaching/comp4161/12s2/slides/week01A/)
text {*
Note that free variables (eg x), bound variables (eg λn) and
constants (eg Suc) are displayed differently. *}
term "x"
term "Suc x"
term "Succ x"
term "Suc x = Succ y"
term "λx
constant "Nat.Suc"
:: nat ⇒ nat
text {* To display more types inside terms: *}
declare [[show_types]]
term "Suc x = Succ y"
text {* To switch off again: *}
declare [[show_types=false]]
term "Suc x = Succ y"
text {* 0 and + are overloaded: *}
prop "n + n = 0"
Sidekick
100%
Tracing
Auto update
Update
"Suc x"
:: "nat"
Console
Output
Prover Session
Theory File
Isabelle Output

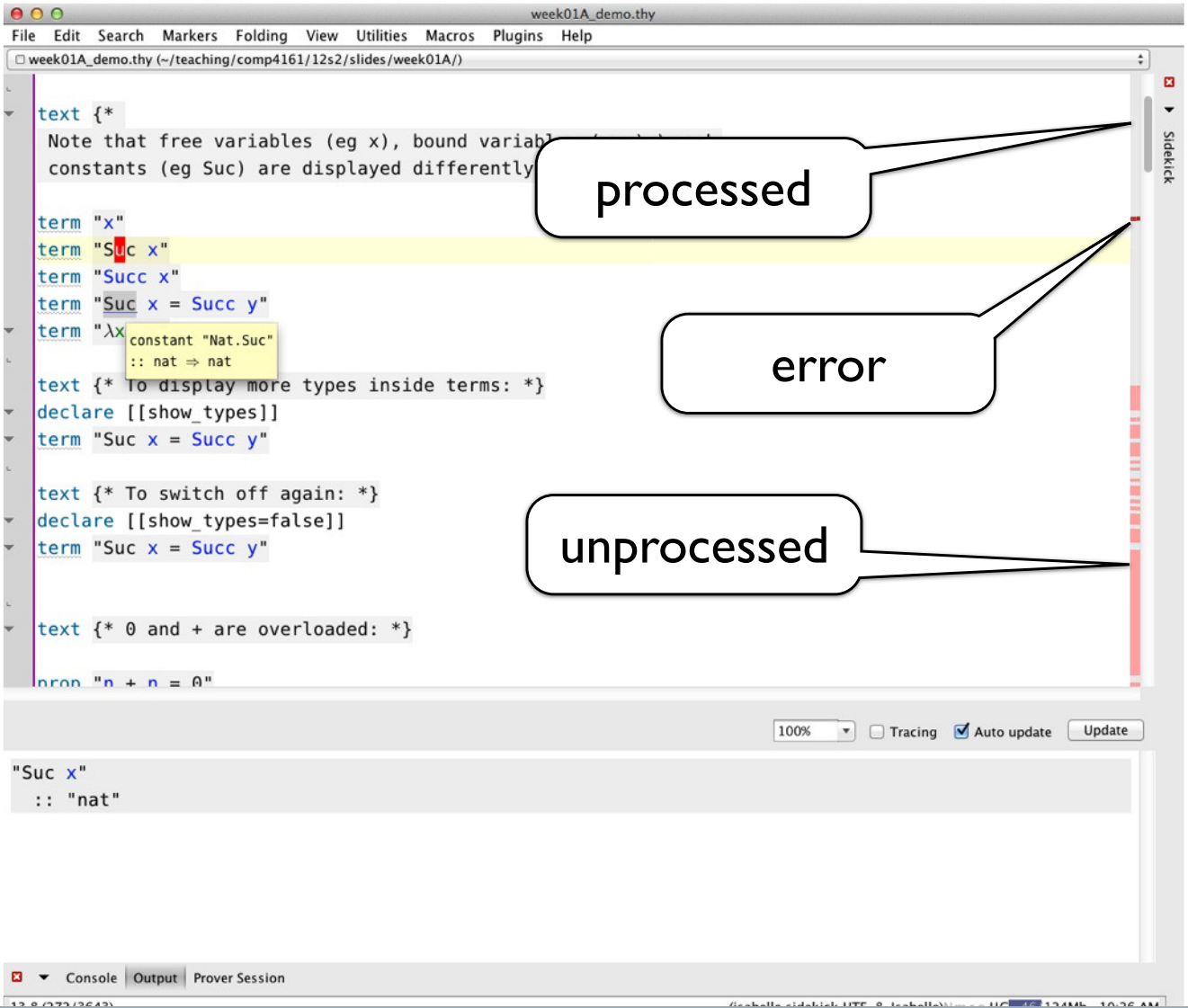
jEdit/PIDE


week01A_demo.thy
File Edit Search Markers Folding View Utilities Macros Plugins Help
week01A_demo.thy (~/teaching/comp4161/12s2/slides/week01A/)
text {*
Note that free variables (eg x), bound variables (eg λn) and
constants (eg Suc) are displayed differently. *}
term "x"
term "Suc x"
term "Succ x"
term "Suc x = Succ y"
term "λx
constant "Nat.Suc"
:: nat ⇒ nat
text {* To display more types insi
declare [[show_types]]
term "Suc x = Succ y"
text {* To switch off again: *}
declare [[show_types=false]]
term "Suc x = Succ y"
text {* 0 and + are overloaded: *}
prop "n + n = 0"
LaTeX Comment
logic terms go in quotes: "x + 2"
Commands
100%
Tracing
Auto update
Update
"Suc x"
:: "nat"
Console
Output
Prover Session

jEdit/PIDE


week01A_demo.thy
File Edit Search Markers Folding View Utilities Macros Plugins Help
week01A_demo.thy (~/teaching/comp4161/12s2/slides/week01A/)
text {*
Note that free variables (eg x), bound variables (eg λn) and
constants (eg Suc) are displayed differently. *}
term "x"
term "Suc x"
term "Succ x"
term "Suc x = Succ y"
constant "Nat.Suc"
:: nat ⇒ nat
text {* To display more types inside terms: *}
declare [[show_types]]
term "Suc x = Succ y"
text {* To switch off again: *}
declare [[show_types=false]]
term "Suc x = Succ y"
text {* 0 and + are overloaded: *}
Command click jumps to definition
Command + hover for popup info
100%
Tracing
Auto update
Update
"Suc x"
:: "nat"
Console
Output
Prover Session

jEdit/PIDE


week01A_demo.thy
File Edit Search Markers Folding View Utilities Macros Plugins Help
week01A_demo.thy (~/teaching/comp4161/12s2/slides/week01A/)
text {*
Note that free variables (eg x), bound variab
constants (eg Suc) are displayed differently
term "x"
term "Suc x"
term "Succ x"
term "Suc x = Succ y"
term "λx
constant "Nat.Suc"
:: nat ⇒ nat
text {* To display more types inside terms: *}
declare [[show_types]]
term "Suc x = Succ y"
text {* To switch off again: *}
declare [[show_types=false]]
term "Suc x = Succ y"
text {* 0 and + are overloaded: *}
prop "n + n = 0"
Sidekick
processed
error
unprocessed
100%
Tracing
Auto update
Update
"Suc x"
:: "nat"
Console
Output
Prover Session

Exercises

- Download and install Isabelle
- Step through the demo files from the lecture web page
- Write your own theory file, look at some theorems in the library, try 'find_theorems'
- How many theorems can help you if you need to prove something containing the term "Suc(Suc x)"?
- What is the name of the theorem for associativity of addition of natural numbers in the library?

Section 3

λ-Calculus

λ-calculus

Alonzo Church

- lived 1903–1995
- supervised people like Alan Turing, Stephen Kleene
- famous for Church-Turing thesis, lambda calculus, first undecidability results
- invented λ calculus in 1930's

λ-calculus

- originally meant as foundation of mathematics
- important applications in theoretical computer science
- foundation of computability and functional programming

untyped λ-calculus

- Turing-complete model of computation
- a simple way of writing down functions

Basic intuition:

instead of	$f(x) = x + 5$
write	$f = \lambda x.\ x + 5$

$\lambda x.\ x + 5$

- a term
- a nameless function
- that adds 5 to its parameter

Function Application

For applying arguments to functions

instead of $f(a)$
write $f\ a$

Example: $(\lambda x.\ x + 5)\ a$

Evaluating: in $(\lambda x.\ t)\ a$ replace x by a in t
(computation!)

Example: $(\lambda x.\ x + 5)\ (a + b)$ evaluates to $(a + b) + 5$

Now Formal

Syntax

Terms: $t \;::=\; v \mid c \mid (t\ t) \mid (\lambda x.\ t)$

$v, x \in V, \quad c \in C, \quad V, C$ sets of names

- v, x variables
- c constants
- $(t\ t)$ application
- $(\lambda x.\ t)$ abstraction

Conventions

- leave out parentheses where possible
- list variables instead of multiple λ

Example: instead of $(\lambda y.\ (\lambda x.\ (x\ y)))$ write $\lambda y\ x.\ x\ y$

Rules:

- list variables: $\lambda x.\ (\lambda y.\ t) = \lambda x\ y.\ t$
- application binds to the left: $x\ y\ z = (x\ y)\ z \neq x\ (y\ z)$
- abstraction binds to the right: $\lambda x.\ x\ y = \lambda x.\ (x\ y) \neq (\lambda x.\ x)\ y$
- leave out outermost parentheses

Getting used to the Syntax

Example:

$\lambda x\ y\ z.\ x\ z\ (y\ z) =$

$\lambda x\ y\ z.\ (x\ z)\ (y\ z) =$

$\lambda x\ y\ z.\ ((x\ z)\ (y\ z)) =$

$\lambda x.\ \lambda y.\ \lambda z.\ ((x\ z)\ (y\ z)) =$

$(\lambda x.\ (\lambda y.\ (\lambda z.\ ((x\ z)\ (y\ z)))))$

Computation

Intuition: replace parameter by argument
this is called β-reduction

Remember: $(\lambda x.\ t)\ a$ is evaluated (noted $\longrightarrow_\beta$) to
t where x is replaced by a

Example:

$(\lambda x\ y.\ \mathit{Suc}\ x\ =\ y)\ 3 \longrightarrow_\beta$
$(\lambda x.\ (\lambda y.\ \mathit{Suc}\ x\ =\ y))\ 3 \longrightarrow_\beta$
$(\lambda y.\ \mathit{Suc}\ 3\ =\ y)$

$(\lambda x\ y.\ f\ (y\ x))\ \ 5\ \ (\lambda x.\ x) \longrightarrow_\beta$
$(\lambda y.\ f\ (y\ 5))\ \ (\lambda x.\ x) \longrightarrow_\beta$
$f\ ((\lambda x.\ x)\ 5) \longrightarrow_\beta$
$f\ 5$

Defining Computation

β reduction:

$$
\begin{array}{lllll}
 & & & & (\lambda x.\ s)\ t \longrightarrow_\beta s[x \leftarrow t] \\
s \longrightarrow_\beta s' & \Longrightarrow & (s\ t) \longrightarrow_\beta (s'\ t) \\
t \longrightarrow_\beta t' & \Longrightarrow & (s\ t) \longrightarrow_\beta (s\ t') \\
s \longrightarrow_\beta s' & \Longrightarrow & (\lambda x.\ s) \longrightarrow_\beta (\lambda x.\ s')
\end{array}
$$

Still to do: define $s[x \leftarrow t]$

Defining Substitution

Easy concept. Small problem: variable capture.
Example: $(\lambda x.\ x\ z)[z \leftarrow x]$

We do **not** want: $(\lambda x.\ x\ x)$ as result. What do we want?

In $(\lambda y.\ y\ z)\ [z \leftarrow x] = (\lambda y.\ y\ x)$ there would be no problem.

So, solution is: rename bound variables.

Free Variables

Bound variables: in $(\lambda x.\ t)$, x is a bound variable.

Free variables FV of a term:

$$\begin{aligned} FV\ (x) &= \{x\} \\ FV\ (c) &= \{\} \\ FV\ (s\ t) &= FV(s) \cup FV(t) \\ FV\ (\lambda x.\ t) &= FV(t) \setminus \{x\} \end{aligned}$$

Example: $FV(\lambda x.\ (\lambda y.\ (\lambda x.\ x)\ y)\ y\ x) = \{y\}$

Term t is called **closed** if $FV(t) = \{\}$

The substitution example, $(\lambda x.\ x\ z)[z \leftarrow x]$, is problematic because the bound variable x is a free variable of the replacement term "x".

Substitution

$$
\begin{array}{lll}
x\,[x \leftarrow t] & = t & \\
y\,[x \leftarrow t] & = y & \text{if } x \neq y \\
c\,[x \leftarrow t] & = c & \\
 & & \\
(s_1\ s_2)\,[x \leftarrow t] & = (s_1[x \leftarrow t]\ s_2[x \leftarrow t]) & \\
 & & \\
(\lambda x.\ s)\,[x \leftarrow t] & = (\lambda x.\ s) & \\
(\lambda y.\ s)\,[x \leftarrow t] & = (\lambda y.\ s[x \leftarrow t]) & \text{if } x \neq y \text{ and } y \notin FV(t) \\
(\lambda y.\ s)\,[x \leftarrow t] & = (\lambda z.\ s[y \leftarrow z][x \leftarrow t]) & \text{if } x \neq y \\
 & & \text{and } z \notin FV(t) \cup FV(s)
\end{array}
$$

Substitution Example

$$
\begin{array}{ll}
 & (x\ \ (\lambda x.\ x)\ \ (\lambda y.\ z\ x))[x \leftarrow y] \\
= & (x[x \leftarrow y])\ \ ((\lambda x.\ x)[x \leftarrow y])\ \ ((\lambda y.\ z\ x)[x \leftarrow y]) \\
= & y\ (\lambda x.\ x)\ (\lambda y'.\ z\ y)
\end{array}
$$

α Conversion

Bound names are irrelevant:
$\lambda x.\ x$ and $\lambda y.\ y$ denote the same function.

α conversion:
$s =_\alpha t$ means $s = t$ up to renaming of bound variables.

Formally:

$$
\begin{array}{lllll}
 & & (\lambda x.\ t) & \longrightarrow_\alpha & (\lambda y.\ t[x \leftarrow y]) \text{ if } y \notin FV(t) \\
s \longrightarrow_\alpha s' & \Longrightarrow & (s\ t) & \longrightarrow_\alpha & (s'\ t) \\
t \longrightarrow_\alpha t' & \Longrightarrow & (s\ t) & \longrightarrow_\alpha & (s\ t') \\
s \longrightarrow_\alpha s' & \Longrightarrow & (\lambda x.\ s) & \longrightarrow_\alpha & (\lambda x.\ s')
\end{array}
$$

$$s =_\alpha t \quad \text{iff} \quad s \longrightarrow_\alpha^* t$$

($\longrightarrow_\alpha^*$ = transitive, reflexive closure of $\longrightarrow_\alpha$ = multiple steps)

α Conversion

Equality in Isabelle is equality modulo α conversion:

if $s =_\alpha t$ then s and t are syntactically equal.

Examples:

$$
\begin{array}{ll}
 & x\ (\lambda x\ y.\ x\ y) \\
=_\alpha & x\ (\lambda y\ x.\ y\ x) \\
=_\alpha & x\ (\lambda z\ y.\ z\ y) \\
\neq_\alpha & z\ (\lambda z\ y.\ z\ y) \\
\neq_\alpha & x\ (\lambda x\ x.\ x\ x)
\end{array}
$$

Back to β

We have defined β reduction: $\longrightarrow_\beta$
Some notation and concepts:

- β **conversion:** $s =_\beta t$ iff $\exists n.\ s \longrightarrow_\beta^* n \wedge t \longrightarrow_\beta^* n$
- t is **reducible** if there is an s such that $t \longrightarrow_\beta s$
- $(\lambda x.\ s)\ t$ is called a **redex** (reducible expression)
- t is reducible iff it contains a redex
- if it is not reducible, t is in **normal form**

Does every λ-term have a normal form?

No!

Example:

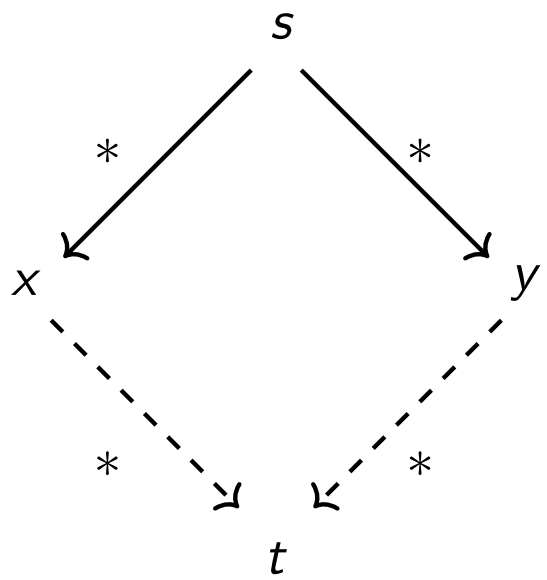
$$(\lambda x.\ x\ x)\ (\lambda x.\ x\ x) \longrightarrow_\beta$$
$$(\lambda x.\ x\ x)\ (\lambda x.\ x\ x) \longrightarrow_\beta$$
$$(\lambda x.\ x\ x)\ (\lambda x.\ x\ x) \longrightarrow_\beta \ldots$$

(but: $(\lambda x\ y.\ y)\ ((\lambda x.\ x\ x)\ (\lambda x.\ x\ x)) \longrightarrow_\beta \lambda y.\ y$)

λ calculus is not terminating

β reduction is confluent

Confluence: $s \longrightarrow_\beta^* x \wedge s \longrightarrow_\beta^* y \Longrightarrow \exists t.\ x \longrightarrow_\beta^* t \wedge y \longrightarrow_\beta^* t$



Order of reduction does not matter for result
Normal forms in λ calculus are unique

β reduction is confluent

Example:

$(\lambda x\ y.\ y)\ ((\lambda x.\ x\ x)\ a) \longrightarrow_\beta (\lambda x\ y.\ y)\ (a\ a) \longrightarrow_\beta \lambda y.\ y$

$(\lambda x\ y.\ y)\ ((\lambda x.\ x\ x)\ a) \longrightarrow_\beta \lambda y.\ y$

η Conversion

Another case of trivially equal functions: $t = (\lambda x.\ t\ x)$
Definition:

$$(\lambda x.\ t\ x) \longrightarrow_\eta t \quad \text{if } x \notin FV(t)$$

$$\frac{s \longrightarrow_\eta s'}{(s\ t) \longrightarrow_\eta (s'\ t)} \qquad \frac{t \longrightarrow_\eta t'}{(s\ t) \longrightarrow_\eta (s\ t')} \qquad \frac{s \longrightarrow_\eta s'}{(\lambda x.\ s) \longrightarrow_\eta (\lambda x.\ s')}$$

$$s =_\eta t \quad \text{iff} \quad \exists n.\ s \longrightarrow_\eta^* n \wedge t \longrightarrow_\eta^* n$$

Example: $(\lambda x.\ f\ x)\ (\lambda y.\ g\ y) \longrightarrow_\eta (\lambda x.\ f\ x)\ g \longrightarrow_\eta f\ g$

- η reduction is confluent and terminating.
- $\longrightarrow_{\beta\eta}$ is confluent.
  $\longrightarrow_{\beta\eta}$ means $\longrightarrow_\beta$ and $\longrightarrow_\eta$ steps are both allowed.
- Equality in Isabelle is also modulo η conversion.

In fact ...

Equality in Isabelle is modulo α, β, and η conversion.

We will see later why that is possible.

Isabelle Demo

So, what can you do with λ calculus?

λ calculus is very expressive, you can encode:

- logic, set theory
- turing machines, functional programs, etc.

Examples:

$$\texttt{true} \equiv \lambda x\ y.\ x \qquad \texttt{if true}\ x\ y \longrightarrow_\beta^* x$$
$$\texttt{false} \equiv \lambda x\ y.\ y \qquad \texttt{if false}\ x\ y \longrightarrow_\beta^* y$$
$$\texttt{if} \equiv \lambda z\ x\ y.\ z\ x\ y$$

Now, `not`, `and`, `or`, etc is easy:

$$\texttt{not} \equiv \lambda x.\ \texttt{if}\ x\ \texttt{false}\ \texttt{true}$$
$$\texttt{and} \equiv \lambda x\ y.\ \texttt{if}\ x\ y\ \texttt{false}$$
$$\texttt{or} \equiv \lambda x\ y.\ \texttt{if}\ x\ \texttt{true}\ y$$

More Examples

Encoding natural numbers (Church Numerals)

$$
\begin{aligned}
0 &\equiv \lambda f\ x.\ x \\
1 &\equiv \lambda f\ x.\ f\ x \\
2 &\equiv \lambda f\ x.\ f\ (f\ x) \\
3 &\equiv \lambda f\ x.\ f\ (f\ (f\ x)) \\
&\ldots
\end{aligned}
$$

Numeral n takes arguments f and x, applies f n-times to x.

$$
\begin{aligned}
\texttt{iszero} &\equiv \lambda n.\ n\ (\lambda x.\ \texttt{false})\ \texttt{true} \\
\texttt{succ} &\equiv \lambda n\ f\ x.\ f\ (n\ f\ x) \\
\texttt{add} &\equiv \lambda m\ n.\ \lambda f\ x.\ m\ f\ (n\ f\ x)
\end{aligned}
$$

Fix Points

$(\lambda x\ f.\ f\ (x\ x\ f))\ \ (\lambda x\ f.\ f\ (x\ x\ f))\ \ t \longrightarrow_\beta$
$(\lambda f.\ f\ ((\lambda x\ f.\ f\ (x\ x\ f))\ \ (\lambda x\ f.\ f\ (x\ x\ f))\ f))\ \ t \longrightarrow_\beta$
$t\ \ ((\lambda x\ f.\ f\ (x\ x\ f))\ (\lambda x\ f.\ f\ (x\ x\ f))\ t)$

$\mu = (\lambda x\ f.\ f\ (x\ x\ f))\ (\lambda x f.\ f\ (x\ x\ f))$
$\mu\ t \longrightarrow_\beta t\ (\mu\ t) \longrightarrow_\beta t\ (t\ (\mu\ t)) \longrightarrow_\beta t\ (t\ (t\ (\mu\ t))) \longrightarrow_\beta \ldots$

$(\lambda x f.\ f\ (x\ x\ f))\ (\lambda x f.\ f\ (x\ x\ f))$ is Turing's fix point operator

Nice, but ...

As a mathematical foundation, λ does not work.
It resulted in an inconsistent logic.

- **Frege** (Predicate Logic, $\sim$ 1879):
  allows arbitrary quantification over predicates
- **Russell** (1901): Paradox $R \equiv \{X | X \notin X\}$
- **Whitehead & Russell** (Principia Mathematica, 1910-1913):
  Fix the problem
- **Church** (1930): λ calculus as logic, `true`, `false`, $\wedge$, ... as λ terms

Problem:

with	$\{x \mid P\ x\} \equiv \lambda x.\ P\ x$	$x \in M \equiv M\ x$
you can write	$R \equiv \lambda x.\ \texttt{not}\ (x\ x)$	
and get	$(R\ R) =_\beta \texttt{not}\ (R\ R)$	
because	$(R\ R) = (\lambda x.\ \texttt{not}\ (x\ x))\ R \longrightarrow_\beta \texttt{not}\ (R\ R)$	

We have learned so far...

- λ calculus syntax
- free variables, substitution
- β reduction
- α and η conversion
- β reduction is confluent
- λ calculus is very expressive (Turing complete)
- λ calculus results in an inconsistent logic

Section 4

Simple-Typed λ-Calculus

λ calculus is inconsistent

Can find term R such that $R\ R\ =_\beta\ \texttt{not}(R\ R)$

There are more terms that do not make sense:

$$1\ 2, \quad \texttt{true}\ \texttt{false}, \quad \text{etc.}$$

Solution: rule out ill-formed terms by using types.
(Church 1940)

Introducing types

Idea: assign a type to each "sensible" λ term.

Examples:

- for *term* t *has type* α write $t :: \alpha$
- if x has type α then $\lambda x.\ x$ is a function from α to α
  Write: $(\lambda x.\ x) :: \alpha \Rightarrow \alpha$
- for $s\ t$ to be sensible:
  s must be a function
  t must be right type for parameter

  If $s :: \alpha \Rightarrow \beta$ and $t :: \alpha$ then $(s\ t) :: \beta$

Now formally again

Syntax for $\lambda^{\to}$

Terms: $t \quad ::= \quad v \mid c \mid (t\ t) \mid (\lambda x.\ t)$

$v, x \in V, \quad c \in C, \quad V, C$ sets of names

Types: $\tau \quad ::= \quad \mathtt{b} \mid \nu \mid \tau \Rightarrow \tau$

$\mathtt{b} \in \{\mathtt{bool}, \mathtt{int}, \ldots\}$ base types

$\nu \in \{\alpha, \beta, \ldots\}$ type variables

$$\alpha \Rightarrow \beta \Rightarrow \gamma \quad = \quad \alpha \Rightarrow (\beta \Rightarrow \gamma)$$

Context Γ:

Γ: function from variable and constant names to types.

Term t has type τ in context Γ: $\quad \Gamma \vdash t :: \tau$

Examples

$\Gamma \vdash (\lambda x.\ x) :: \alpha \Rightarrow \alpha$

$[y \leftarrow \texttt{int}] \vdash y :: \texttt{int}$

$[z \leftarrow \texttt{bool}] \vdash (\lambda y.\ y)\ z :: \texttt{bool}$

$[] \vdash \lambda f\ x.\ f\ x :: (\alpha \Rightarrow \beta) \Rightarrow \alpha \Rightarrow \beta$

A term t is **well typed** or **type correct**
if there are Γ and τ such that $\Gamma \vdash t :: \tau$

Type Checking Rules

Variables:

$$\frac{}{\Gamma \vdash x :: \Gamma(x)}$$

Application:

$$\frac{\Gamma \vdash t_1 :: \tau_2 \Rightarrow \tau \quad \Gamma \vdash t_2 :: \tau_2}{\Gamma \vdash (t_1\ t_2) :: \tau}$$

Abstraction:

$$\frac{\Gamma[x \leftarrow \tau_x] \vdash t :: \tau}{\Gamma \vdash (\lambda x.\ t) :: \tau_x \Rightarrow \tau}$$

Example Type Derivation

$$\dfrac{\dfrac{\dfrac{}{[x \leftarrow \alpha, y \leftarrow \beta] \vdash x :: \alpha}\,\mathit{Var}}{[x \leftarrow \alpha] \vdash \lambda y.\, x :: \beta \Rightarrow \alpha}\,\mathit{Abs}}{[] \vdash \lambda x\ y.\, x :: \alpha \Rightarrow \beta \Rightarrow \alpha}\,\mathit{Abs}$$

Remember:

$$\frac{}{\Gamma \vdash x :: \Gamma(x)}\,\mathit{Var} \qquad \frac{\Gamma \vdash t_1 :: \tau_2 \Rightarrow \tau \quad \Gamma \vdash t_2 :: \tau_2}{\Gamma \vdash (t_1\ t_2) :: \tau}\,\mathit{App} \qquad \frac{\Gamma[x \leftarrow \tau_x] \vdash t :: \tau}{\Gamma \vdash (\lambda x.\, t) :: \tau_x \Rightarrow \tau}\,\mathit{Abs}$$

More complex Example

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma \vdash f :: \alpha \Rightarrow (\alpha \Rightarrow \beta)}\,Var \quad \dfrac{}{\Gamma \vdash x :: \alpha}\,Var}{\Gamma \vdash f\ x :: \alpha \Rightarrow \beta}\,App \quad \dfrac{}{\Gamma \vdash x :: \alpha}\,Var}{\Gamma \vdash f\ x\ x :: \beta}\,App}{[f \leftarrow \alpha \Rightarrow \alpha \Rightarrow \beta] \vdash \lambda x.\ f\ x\ x :: \alpha \Rightarrow \beta}\,Abs}{[] \vdash \lambda f\ x.\ f\ x\ x :: (\alpha \Rightarrow \alpha \Rightarrow \beta) \Rightarrow \alpha \Rightarrow \beta}\,Abs$$

$$\Gamma = [f \leftarrow \alpha \Rightarrow \alpha \Rightarrow \beta, x \leftarrow \alpha]$$

Remember:

$$\dfrac{}{\Gamma \vdash x :: \Gamma(x)}\,Var \qquad \dfrac{\Gamma \vdash t_1 :: \tau_2 \Rightarrow \tau \quad \Gamma \vdash t_2 :: \tau_2}{\Gamma \vdash (t_1\ t_2) :: \tau}\,App \qquad \dfrac{\Gamma[x \leftarrow \tau_x] \vdash t :: \tau}{\Gamma \vdash (\lambda x.\ t) :: \tau_x \Rightarrow \tau}\,Abs$$

More general Types

- A term can have more than one type.

  Example: $[] \vdash \lambda x.\ x :: \texttt{bool} \Rightarrow \texttt{bool}$
  $[] \vdash \lambda x.\ x :: \alpha \Rightarrow \alpha$

- Some types are more general than others:

  $\tau \lesssim \sigma$ if there is a substitution S such that $\tau = S(\sigma)$

  Examples:

  $$\texttt{int} \Rightarrow \texttt{bool} \quad \lesssim \quad \alpha \Rightarrow \beta \quad \lesssim \quad \beta \Rightarrow \alpha \quad \not\lesssim \quad \alpha \Rightarrow \alpha$$

Most general Types

Fact: each type correct term has a most general type

Formally:

$$\Gamma \vdash t :: \tau \quad \Longrightarrow \quad \exists \sigma.\, \Gamma \vdash t :: \sigma \wedge (\forall \sigma'.\, \Gamma \vdash t :: \sigma' \Longrightarrow \sigma' \lesssim \sigma)$$

It can be found by executing the typing rules backwards.

- **type checking:** checking if $\Gamma \vdash t :: \tau$ for given Γ and τ
- **type inference:** computing Γ and τ such that $\Gamma \vdash t :: \tau$

Type checking and type inference on $\lambda^{\rightarrow}$ are decidable.

What about β reduction?

Definition of β reduction stays the same.

Fact: Well typed terms stay well typed during β reduction

Formally: $\Gamma \vdash s :: \tau \ \wedge \ s \longrightarrow_\beta t \Longrightarrow \Gamma \vdash t :: \tau$

This property is called **subject reduction**

What about termination?

β reduction in $\lambda^{\rightarrow}$ always terminates.

(Alan Turing, 1942)

- $=_\beta$ is decidable
  To decide if $s =_\beta t$, reduce s and t to normal form (always exists, because $\longrightarrow_\beta$ terminates), and compare result.
- $=_{\alpha\beta\eta}$ is decidable
  This is why Isabelle can automatically reduce each term to $\beta\eta$ normal form.

What does this mean for Expressiveness?

Checkpoint:

- untyped lambda calculus is Turing complete
  (all computable functions can be expressed)
- but it is inconsistent
- $\lambda^{\rightarrow}$ "fixes" the inconsistency problem by adding types
- Problem: it is not Turing complete anymore!

Not all computable functions can be expressed in $\lambda^{\rightarrow}$!
(non terminating functions cannot be expressed)

But wait... typed functional languages are turing complete!

What does this mean for Expressiveness?

So...

- typed functional languages are turing complete
- but $\lambda^{\rightarrow}$ is not...
- How does this work?
- By adding one single constant, the Y operator (fix point operator), to $\lambda^{\rightarrow}$
- This introduces the non-termination that the types removed.

$$Y :: (\tau \Rightarrow \tau) \Rightarrow \tau$$
$$Y\ t \longrightarrow_{\beta} t\ (Y\ t)$$

Fact: If we add Y to $\lambda^{\rightarrow}$ as the only constant, then each computable function can be encoded as closed, type correct $\lambda^{\rightarrow}$ term.

- Y is used for recursion
- lose decidability (what does $Y\ (\lambda x.\ x)$ reduce to?)
- (Isabelle/HOL doesn't have Y; recursion is more restricted)

Types and Terms in Isabelle

Types: $\tau \;::=\; \mathtt{b} \mid {'\nu} \mid {'\nu :: C} \mid \tau \Rightarrow \tau \mid (\tau, \ldots, \tau)\, K$

$\mathtt{b} \in \{\mathtt{bool}, \mathtt{int}, \ldots\}$ base types

$\nu \in \{\alpha, \beta, \ldots\}$ type variables

$K \in \{\mathtt{set}, \mathtt{list}, \ldots\}$ type constructors

$C \in \{\mathtt{order}, \mathtt{linord}, \ldots\}$ type classes

Terms: $t \;::=\; v \mid c \mid ?v \mid (t\ t) \mid (\lambda x.\ t)$

$v, x \in V, \quad c \in C, \quad V, C$ sets of names

- **type constructors**: construct a new type out of a parameter type. Example: `int list`
- **type classes**: restrict type variables to a class defined by axioms. Example: $\alpha :: \mathit{order}$
- **schematic variables**: variables that can be instantiated.

Type Classes

- similar to Haskell's type classes, but with semantic properties

  class order $=$
  assumes order_refl: "$x \leq x$"
  assumes order_trans: "$[\![x \leq y; y \leq z]\!] \Longrightarrow x \leq z$"
  ...

- theorems can be proved in the abstract

  lemma order_less_trans: "$\bigwedge x ::'a :: \mathit{order}.\ [\![x < y; y < z]\!] \Longrightarrow x < z$"

- can be used for subtyping

  class linorder $=$ order $+$
  assumes linorder_linear: "$x \leq y \vee y \leq x$"

- can be instantiated

  instance nat :: "{order, linorder}" **by** ...

Schematic Variables

$$\frac{X \quad Y}{X \wedge Y}$$

- X and Y must be instantiated to apply the rule

But: **lemma** "$x + 0 = 0 + x$"

- x is free
- convention: lemma must be true for all x
- **during the proof**, x must **not** be instantiated

Solution:
Isabelle has free (x), bound (x), and schematic (?X) variables.

Only schematic variables can be instantiated.

Free converted into schematic after proof is finished.

Higher Order Unification

Unification:
Find substitution σ on variables for terms s, t such that $\sigma(s) = \sigma(t)$

In Isabelle:
Find substitution σ on schematic variables such that $\sigma(s) =_{\alpha\beta\eta} \sigma(t)$

Examples:

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

Higher Order: schematic variables can be functions.

Higher Order Unification

- Unification modulo $\alpha\beta$ (Higher Order Unification) is semi-decidable
- Unification modulo $\alpha\beta\eta$ is undecidable
- Higher Order Unification has possibly infinitely many solutions

But:

- Most cases are well-behaved
- Important fragments (like Higher Order Patterns) are decidable

Higher Order Pattern:

- is a term in β normal form where
- each occurrence of a schematic variable is of the form $?f\ t_1\ \ldots\ t_n$
- and the $t_1\ \ldots\ t_n$ are η-convertible into n distinct bound variables

We have learned so far...

- Simply typed lambda calculus: $\lambda^{\rightarrow}$
- Typing rules for $\lambda^{\rightarrow}$, type variables, type contexts
- β-reduction in $\lambda^{\rightarrow}$ satisfies subject reduction
- β-reduction in $\lambda^{\rightarrow}$ always terminates
- Types and terms in Isabelle

Exercises

- Construct a type derivation tree for the term $\lambda x\ y\ z.\ z\ x\ (y\ x)$
- Find a unifier (substitution) such that
  $\lambda x\ y\ z.\ ?F\ y\ z = \lambda x\ y\ z.\ z\ (?G\ x\ y)$

Section 5

Isabelle/HOL
Natural Deduction

Preview: Proofs in Isabelle

Proofs in Isabelle

General schema:

lemma name: ”<goal>”
apply <method>
apply <method>
...
done

- Sequential application of methods until all **subgoals** are solved.

The Proof State

1. $\bigwedge x_1 \ldots x_p.[\![A_1; \ldots; A_n]\!] \Longrightarrow B$
2. $\bigwedge y_1 \ldots y_q.[\![C_1; \ldots; C_m]\!] \Longrightarrow D$

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

Isabelle Theories

Syntax:

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

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

Unless you need something special:

`theory` *MyTh* `imports Main begin ... end`

Natural Deduction Rules

$$\frac{A \quad B}{A \wedge B}\ \text{conjI} \qquad \frac{A \wedge B \quad [\![A; B]\!] \Longrightarrow C}{C}\ \text{conjE}$$

$$\frac{A}{A \vee B}\ \frac{B}{A \vee B}\ \text{disjI1/2} \qquad \frac{A \vee B \quad A \Longrightarrow C \quad B \Longrightarrow C}{C}\ \text{disjE}$$

$$\frac{A \Longrightarrow B}{A \longrightarrow B}\ \text{impI} \qquad \frac{A \longrightarrow B \quad A \quad B \Longrightarrow C}{C}\ \text{impE}$$

For each connective ($\wedge$, $\vee$, etc):
introduction and **elimination** rules

Proof by Assumption

apply assumption

proves

1. $[\![B_1; \ldots; B_m]\!] \Longrightarrow C$

by unifying C with one of the B_i

There may be more than one matching B_i and multiple unifiers.

Backtracking!

Explicit backtracking command: **back**

Intro Rules

Intro rules decompose formulae to the right of $\Longrightarrow$.

apply (rule <intro-rule>)

Intro rule $[\![A_1; \ldots; A_n]\!] \Longrightarrow A$ means

- To prove A it suffices to show $A_1 \ldots A_n$

Applying rule $[\![A_1; \ldots; A_n]\!] \Longrightarrow A$ to subgoal C:

- unify A and C
- replace C with n new subgoals $A_1 \ldots A_n$

Intro Rules: example

To prove subgoal $A \longrightarrow A$ we can use: $\dfrac{P \Longrightarrow Q}{P \longrightarrow Q}$ impI

(in Isabelle: *impI* : $(?P \Longrightarrow ?Q) \Longrightarrow ?P \longrightarrow ?Q$)

Recall:

Applying rule $[\![A_1; \ldots; A_n]\!] \Longrightarrow A$ to subgoal C:

- unify A and C
- replace C with n new subgoals $A_1 \ldots A_n$

Here:

- unify... $?P \longrightarrow ?Q$ with $A \longrightarrow A$
- replace subgoal... $A \longrightarrow A$ (i.e. $[\![\,]\!] \Longrightarrow A \longrightarrow A$)
  with $[\![\,A\,]\!] \Longrightarrow A$ (which can be proved with: **apply** assumption)

Elim Rules

Elim rules decompose formulae on the left of $\Longrightarrow$.

apply (erule <elim-rule>)

Elim rule $[\![A_1; \ldots; A_n]\!] \Longrightarrow A$ means

- If I know A_1 and want to prove A it suffices to show A_2 ... A_n

Applying rule $[\![A_1; \ldots; A_n]\!] \Longrightarrow A$ to subgoal C:
Like **rule** but also

- unifies first premise of rule with an assumption
- eliminates that assumption

Elim Rules: example

To prove $[\![B \wedge A]\!] \Longrightarrow A$ we can use: $\dfrac{P \wedge Q \quad [\![P; Q]\!] \Longrightarrow R}{R}$ conjE

(in Isabelle: $conjE : [\![?P \wedge ?Q;\ [\![?P;\ ?Q]\!] \Longrightarrow ?R]\!] \Longrightarrow ?R$)

Recall:

Applying rule $[\![A_1; \ldots; A_n]\!] \Longrightarrow A$ to subgoal C:
Like **rule** but also

- unifies first premise of rule with an assumption
- eliminates that assumption

Here:

- unify... $?R$ with A
- and also unify... $?P \wedge ?Q$ with assumption $B \wedge A$
- replace subgoal... $[\![B \wedge A]\!] \Longrightarrow A$
  with $[\![B; A]\!] \Longrightarrow A$ (which can be proved with: **apply** assumption)

Demo

More Proof Rules

Iff, Negation, True and False

$$\frac{A \Longrightarrow B \quad B \Longrightarrow A}{A = B}\ \text{iffI} \qquad \frac{A = B \quad [\![A \longrightarrow B; B \longrightarrow A]\!] \Longrightarrow C}{C}\ \text{iffE}$$

$$\frac{A = B}{A \Longrightarrow B}\ \text{iffD1} \qquad \frac{A = B}{B \Longrightarrow A}\ \text{iffD2}$$

$$\frac{A \Longrightarrow \mathit{False}}{\neg A}\ \text{notI} \qquad \frac{\neg A \quad A}{P}\ \text{notE}$$

$$\frac{}{\mathit{True}}\ \text{TrueI} \qquad \frac{\mathit{False}}{P}\ \text{FalseE}$$

Equality

$$\frac{}{t = t}\ \text{refl} \qquad \frac{s = t}{t = s}\ \text{sym} \qquad \frac{r = s \quad s = t}{r = t}\ \text{trans}$$

$$\frac{s = t \quad P\ s}{P\ t}\ \text{subst}$$

Rarely needed explicitly — used implicitly by term rewriting

Classical

$$\frac{}{P = \mathit{True} \vee P = \mathit{False}}\ \text{True-or-False}$$

$$\frac{}{P \vee \neg P}\ \text{excluded-middle}$$

$$\frac{\neg A \Longrightarrow \mathit{False}}{A}\ \text{ccontr} \qquad \frac{\neg A \Longrightarrow A}{A}\ \text{classical}$$

- **excluded-middle**, **ccontr** and **classical** not derivable from the other rules.
- if we include True-or-False, they are derivable

They make the logic "classical", "non-constructive"

Cases

$$\frac{}{P \vee \neg P}\ \text{excluded-middle}$$

is a case distinction on type *bool*

Isabelle can do case distinctions on arbitrary terms:

apply (case_tac *term*)

Safe and not so safe

Safe rules preserve provability

conjI, impI, notI, iffI, refl, ccontr, classical, conjE, disjE

$$\frac{A \quad B}{A \wedge B}\ \text{conjI}$$

Unsafe rules can turn a provable goal into an unprovable one

disjI1, disjI2, impE, iffD1, iffD2, notE

$$\frac{A}{A \vee B}\ \text{disjI1}$$

Apply safe rules before unsafe ones

Demo

What we have learned so far . . .

- natural deduction rules for $\wedge$, $\vee$, $\longrightarrow$, $\neg$, iff...
- proof by assumption, by intro rule, elim rule
- safe and unsafe rules
- indent your proofs! (one space per subgoal)
- prefer implicit backtracking (chaining) or *rule_tac*, instead of *back*
- *prefer* and *defer*
- *oops* and *sorry*

Section 6

Isabelle/HOL
First-Order Logic

Last time...

- natural deduction rules for $\wedge$, $\vee$, $\longrightarrow$, $\neg$, iff...
- proof by assumption, by intro rule, elim rule
- safe and unsafe rules
- indent your proofs! (one space per subgoal)
- prefer implicit backtracking (chaining) or *rule_tac*, instead of *back*
- *prefer* and *defer*
- *oops* and *sorry*

Quantifiers

Scope

- Scope of parameters: whole subgoal
- Scope of $\forall$, $\exists$, ...: ends with ; or $\Longrightarrow$

Example:

$$\bigwedge x\ y.\ [\![\ \forall y.\ P\ y \longrightarrow Q\ z\ y;\ \ Q\ x\ y\ ]\!] \Longrightarrow\ \exists x.\ Q\ x\ y$$

means

$$\bigwedge x\ y.\ [\![\ (\forall y_1.\ P\ y_1 \longrightarrow Q\ z\ y_1);\ \ Q\ x\ y\ ]\!] \Longrightarrow\ (\exists x_1.\ Q\ x_1\ y)$$

Natural deduction for quantifiers

$$\frac{\bigwedge x.\ P\ x}{\forall x.\ P\ x}\ \text{allI} \qquad \frac{\forall x.\ P\ x \quad P\ ?x \Longrightarrow R}{R}\ \text{allE}$$

$$\frac{P\ ?x}{\exists x.\ P\ x}\ \text{exI} \qquad \frac{\exists x.\ P\ x \quad \bigwedge x.\ P\ x \Longrightarrow R}{R}\ \text{exE}$$

- **allI** and **exE** introduce new parameters ($\bigwedge x$).
- **allE** and **exI** introduce new unknowns ($?x$).

Instantiating Rules

apply (rule_tac x = "*term*" in *rule*)

Like **rule**, but ?*x* in *rule* is instantiated by *term* before application.

Similar: **erule_tac**

! *x* **is in** *rule*, **not in goal** !

Two Successful Proofs

1. $\forall x.\ \exists y.\ x = y$

apply (rule alll)

1. $\bigwedge x.\ \exists y.\ x = y$

best practice

apply (rule_tac x = "x" in exI)

1. $\bigwedge x.\ x = x$

apply (rule refl)

simpler & clearer

exploration

apply (rule exI)

1. $\bigwedge x.\ x = ?y\ x$

apply (rule refl)

$?y \mapsto \lambda u.u$

shorter & trickier

Two Unsuccessful Proofs

1. $\exists y.\ \forall x.\ x = y$

apply (rule_tac x = ??? in exI)

apply (rule exI)

1. $\forall x.\ x = ?y$

apply (rule allI)

1. $\bigwedge x.\ x = ?y$

apply (rule refl)

$?y \mapsto x$ yields $\bigwedge x'.\ x' = x$

Principle:

$?f\ x_1 \ldots x_n$ ***can only be replaced by term*** t

if $params(t) \subseteq x_1, \ldots, x_n$

Safe and Unsafe Rules

Safe allI, exE

Unsafe allE, exI

Create parameters first, unknowns later

Demo: Quantifier Proofs

Parameter names

Parameter names are chosen by Isabelle

1. $\forall x.\ \exists y.\ x = y$

   apply (rule allI)

1. $\bigwedge x.\ \exists y.\ x = y$

   apply (rule_tac x = "x" in exI)

Brittle!

Renaming parameters

1. $\forall x.\ \exists y.\ x = y$

   apply (rule allI)

1. $\bigwedge x.\ \exists y.\ x = y$

   apply (rename_tac N)

1. $\bigwedge N.\ \exists y.\ N = y$

   apply (rule_tac x = "N" in exI)

In general:

(rename_tac $x_1 \ldots x_n$**)** renames the rightmost (inner) n parameters to $x_1 \ldots x_n$

Forward Proof: frule and drule

apply (frule $<$ *rule* $>$)

Rule:	$[\![A_1; \ldots; A_m]\!] \Longrightarrow A$
Subgoal:	1. $[\![B_1; \ldots; B_n]\!] \Longrightarrow C$
Substitution:	$\sigma(B_i) \equiv \sigma(A_1)$
New subgoals:	1. $\sigma([\![B_1; \ldots; B_n]\!] \Longrightarrow A_2)$
	$\vdots$
	m-1. $\sigma([\![B_1; \ldots; B_n]\!] \Longrightarrow A_m)$
	m. $\sigma([\![B_1; \ldots; B_n; A]\!] \Longrightarrow C)$

Like **frule** but also deletes B_i: **apply** (drule $<$ *rule* $>$)

Examples for Forward Rules

$$\frac{P \wedge Q}{P}\ \text{conjunct1} \qquad \frac{P \wedge Q}{Q}\ \text{conjunct2}$$

$$\frac{P \longrightarrow Q \quad P}{Q}\ \text{mp}$$

$$\frac{\forall x.\ P\ x}{P\ ?x}\ \text{spec}$$

Forward Proof: OF

$$r\ [\textbf{OF}\ r_1 \ldots r_n]$$

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

Example:

dvd_add : $[\![\ ?a\ dvd\ ?b; ?a\ dvd\ ?c\ ]\!] \Longrightarrow ?a\ dvd\ ?b + ?c$

dvd_refl : $?a\ dvd\ ?a$

dvd_add[**OF** *dvd_refl*] : $[\![\ ?a\ dvd\ ?c\ ]\!] \Longrightarrow ?a\ dvd\ ?a + ?c$

Forward proofs: THEN

r_1 [THEN r_2] means r_2 [OF r_1]

Demo: Forward Proofs

Hilbert's Epsilon Operator

(David Hilbert, 1862-1943)

$\varepsilon\ x.\ Px$ is a value that satisfies P (if such a value exists)

ε also known as **description operator**.
In Isabelle the ε-operator is written SOME $x.\ P\ x$

$$\frac{P\ ?x}{P\ (\text{SOME}\ x.\ P\ x)}\ \text{somel}$$

More Epsilon

ε implies Axiom of Choice:

$$\forall x.\ \exists y.\ Q\ x\ y \Longrightarrow \exists f.\ \forall x.\ Q\ x\ (f\ x)$$

Existential and universal quantification can be defined with ε.

Isabelle also knows the definite description operator **THE** (aka ι):

$$\frac{}{(\mathrm{THE}\ x.\ x = a) = a}\ \text{the\_eq\_trivial}$$

Some Automation

More Proof Methods:

apply (intro <intro-rules>)	repeatedly applies intro rules
apply (elim <elim-rules>)	repeatedly applies elim rules
apply clarify	applies all safe rules that do not split the goal
apply safe	applies all safe rules
apply blast	an automatic tableaux prover (works well on predicate logic)
apply fast	another automatic search tactic

Exercises

- We said that ε implies the Axiom of Choice:

  $\forall x.\ \exists y.\ Q\ x\ y \Longrightarrow \exists f.\ \forall x.\ Q\ x\ (f\ x)$
- Prove the axiom of choice as a lemma, using only the introduction and elimination rules for $\forall$ and $\exists$, namely `allI`, `exI`, `allE`, `exE`, and the introduction rule for ε, `someI`, using only the proof methods `rule`, `rule_tac`, `erule`, `erule_tac` and `assumption`.

We have learned so far...

- Proof rules for predicate calculus
- Safe and unsafe rules
- Forward Proof
- The Epsilon Operator
- Some automation

Section 7

Isabelle/HOL
Isar (Part 1)
A Language for Structured Proofs

Motivation

Is this true: $(A \longrightarrow B) = (B \vee \neg A)$?

Motivation

Is this true: $(A \longrightarrow B) = (B \vee \neg A)$?

YES!

```
apply (rule iffI)
 apply (cases A)
  apply (rule disjI1)
  apply (erule impE)
   apply assumption
  apply assumption
 apply (rule disjI2)
 apply assumption
apply (rule impI)
apply (erule disjE)
 apply assumption
apply (erule notE)
apply assumption
done
```

or `by blast`

OK it's true. But WHY?

Motivation

WHY is this true: $(A \longrightarrow B) = (B \vee \neg A)$?

Demo

Isar

apply scripts

→ hard to read
→ hard to maintain

No explicit structure.

What about..

→ Elegance?
→ Explaining deeper insights?

Isar!

A typical Isar proof

proof
  assume $formula_0$
  have $formula_1$ **by** simp
  ⋮
  have $formula_n$ **by** blast
  show $formula_{n+1}$ **by** ...
qed

proves $formula_0 \Longrightarrow formula_{n+1}$

(analogous to **assumes**/**shows** in lemma statements)

Isar core syntax

proof = **proof** [method] statement* **qed**
| **by** method

method = (simp . . .) | (blast . . .) | (rule . . .) | . . .

statement = **fix** variables ($\bigwedge$)
| **assume** proposition ($\Longrightarrow$)
| [**from** name$^+$] (**have** | **show**) proposition proof
| **next** (separates subgoals)

proposition = [name:] formula

proof and qed

proof [method] statement* **qed**

lemma "$[\![A; B]\!] \Longrightarrow A \wedge B$"
proof (rule conjI)
  assume A: "A"
  from A **show** "A" **by** assumption
next
  assume B: "B"
  from B **show** "B" **by** assumption
qed

$\rightarrow$	**proof** (<method>)	applies method to the stated goal
$\rightarrow$	**proof**	applies a single rule that fits
$\rightarrow$	**proof -**	does nothing to the goal

How do I know what to Assume and Show?

Look at the proof state!

lemma "$[\![A; B]\!] \Longrightarrow A \wedge B$"
proof (rule conjI)

- **proof** (rule conjI) changes proof state to
  1. $[\![A; B]\!] \Longrightarrow A$
  2. $[\![A; B]\!] \Longrightarrow B$
- so we need 2 shows: **show** "A" and **show** "B"
- We are allowed to **assume** A,
  because A is in the assumptions of the proof state.

The Three Modes of Isar

- **[prove]**:
  goal has been stated, proof needs to follow.
- **[state]**:
  proof block has opened or subgoal has been proved,
  new *from* statement, goal statement or assumptions can follow.
- **[chain]**:
  from statement has been made, goal statement needs to follow.

lemma "$[\![A; B]\!] \Longrightarrow A \wedge B$" **[prove]**
proof (rule conjI) **[state]**
  assume A: "A" **[state]**
  from A **[chain]** **show** "A" **[prove]** **by** assumption **[state]**
next **[state]** **...**

Have

Can be used to make intermediate steps.

Example:

lemma "$(x :: \mathsf{nat}) + 1 = 1 + x$"
proof -
  have A: "$x + 1 = \mathsf{Suc}\ x$" **by** simp
  have B: "$1 + x = \mathsf{Suc}\ x$" **by** simp
  show "$x + 1 = 1 + x$" **by** (simp only: A B)
qed

Demo

Backward and Forward

Backward reasoning: ... **have** "$A \wedge B$" **proof**

- **proof** picks an **intro** rule automatically
- conclusion of rule must unify with $A \wedge B$

Forward reasoning: ...

assume AB: "$A \wedge B$"
from AB **have** "..." **proof**

- now **proof** picks an **elim** rule automatically
- triggered by **from**
- first assumption of rule must unify with AB

General case: **from** A_1 ... A_n **have** R **proof**

- first n assumptions of rule must unify with A_1 ... A_n
- conclusion of rule must unify with R

Fix and Obtain

- **fix** $v_1 \ldots v_n$

  Introduces new arbitrary but fixed variables
  ($\sim$ parameters, $\bigwedge$)

- **obtain** $v_1 \ldots v_n$ **where** <prop> <proof>

  Introduces new variables together with property

Fancy Abbreviations

this	=	the previous fact proved or assumed
then	=	**from** this
thus	=	**then show**
hence	=	**then have**
with $A_1 \ldots A_n$	=	**from** $A_1 \ldots A_n$ this
?thesis	=	the last enclosing goal statement

Demo

Moreover and Ultimately

have X_1: P_1 ...	**have** P_1 ...
have X_2: P_2 ...	**moreover have** P_2 ...
⋮	⋮
have X_n: P_n ...	**moreover have** P_n ...
from X_1 ... X_n **show** ...	**ultimately show** ...

wastes lots of brain power on names X_1 ... X_n

General Case Distinctions

show *formula*
proof -
  have $P_1 \vee P_2 \vee P_3$ <proof>
  moreover { **assume** P_1 ... **have** ?thesis <proof> }
  moreover { **assume** P_2 ... **have** ?thesis <proof> }
  moreover { **assume** P_3 ... **have** ?thesis <proof> }
  ultimately show ?thesis **by** blast
qed

{ . . . } is a proof block similar to **proof** ... **qed**

{ **assume** P_1 . . . **have** P <proof> }
stands for $P_1 \Longrightarrow P$

Mixing proof styles

from . . .
have . . .
  apply - make incoming facts assumptions
  apply (. . .)
  ⋮
  apply (. . .)
  done

More on Automation

This can be automated

Automated methods (fast, blast, clarify etc) are not hardwired.
Safe/unsafe intro/elim rules can be declared.

Syntax:

[<kind>!]	for safe rules (<kind> one of intro, elim, dest)
[<kind>]	for unsafe rules

Application (roughly):
do safe rules first, search/backtrack on unsafe rules only

Example:

declare attribute globally	**declare** conjI [intro!] allE [elim]
remove attribute globally	**declare** allE [rule del]
use locally	**apply** (blast intro: someI)
delete locally	**apply** (blast del: conjI)

Demo: Automation

Exercises

- derive the classical contradiction rule $(\neg P \Longrightarrow \mathit{False}) \Longrightarrow P$ in Isabelle
- define **nor** and **nand** in Isabelle
- show $\mathsf{nor}\ x\ x\ =\ \mathsf{nand}\ x\ x$
- derive safe intro and elim rules for them
- use these in an automated proof of $\mathsf{nor}\ x\ x\ =\ \mathsf{nand}\ x\ x$

Section 8

Higher Order Logic

What is Higher Order Logic?

- **Propositional Logic:**
  - no quantifiers
  - all variables have type bool
- **First Order Logic:**
  - quantification over values, but not over functions and predicates,
  - terms and formulas syntactically distinct
- **Higher Order Logic:**
  - quantification over everything, including predicates
  - consistency by types
  - formula = term of type bool
  - definition built on $\lambda^{\rightarrow}$ with certain default types and constants

Defining Higher Order Logic

Default types:

bool $\quad$ _ $\Rightarrow$ _ $\quad$ ind

- **bool** sometimes called *o*
- $\Rightarrow$ sometimes called *fun*

Default Constants:

$$
\begin{array}{lll}
\longrightarrow & :: & \mathit{bool} \Rightarrow \mathit{bool} \Rightarrow \mathit{bool} \\
= & :: & \alpha \Rightarrow \alpha \Rightarrow \mathit{bool} \\
\epsilon & :: & (\alpha \Rightarrow \mathit{bool}) \Rightarrow \alpha
\end{array}
$$

Higher Order Abstract Syntax

Problem: Define syntax for binders like $\forall$, $\exists$, ε

One approach: $\forall :: \mathit{var} \Rightarrow \mathit{term} \Rightarrow \mathit{bool}$
Drawback: need to think about substitution, α conversion again.

But: Already have binder, substitution, α conversion in meta logic

$$\lambda$$

So: Use λ to encode all other binders.

Higher Order Abstract Syntax

Example:

$$\text{ALL} :: (\alpha \Rightarrow \mathit{bool}) \Rightarrow \mathit{bool}$$

HOAS	**usual syntax**
ALL $(\lambda x.\ x = 2)$	$\forall x.\ x = 2$
ALL P	$\forall x.\ P\ x$

Isabelle can translate usual binder syntax into HOAS.

Side Track: Syntax Declarations

- **mixfix:**
  consts drvbl :: $ct \Rightarrow ct \Rightarrow fm \Rightarrow bool$ ("_, _ ⊢ _")
  Legal syntax now: $\Gamma, \Pi \vdash F$
- **priorities:**
  pattern can be annotated with priorities to indicate binding strength
  Example: drvbl :: $ct \Rightarrow ct \Rightarrow fm \Rightarrow bool$ ("_, _ ⊢ _" [30, 0, 20] 60)
- **infixl/infixr**: short form for left/right associative binary operators
  Example: or :: $bool \Rightarrow bool \Rightarrow bool$ (infixr "∨" 30)
- **binders:** declaration must be of the form
  $c :: (\tau_1 \Rightarrow \tau_2) \Rightarrow \tau_3$ (binder "B" $< p >$)
  $B\ x.\ P\ x$ translated into $c\ P$ (and vice versa)
  Example ALL :: $(\alpha \Rightarrow bool) \Rightarrow bool$ (binder "∀" 10)

More in Isabelle/Isar Reference Manual (8.2)

Back to HOL

Base: $bool, \Rightarrow, ind \qquad =, \longrightarrow, \varepsilon$

And the rest is definitions:

$$
\begin{array}{lcl}
\text{True} & \equiv & (\lambda x :: bool.\ x) = (\lambda x.\ x) \\
\text{All } P & \equiv & P = (\lambda x.\ \text{True}) \\
\text{Ex } P & \equiv & \forall Q.\ (\forall x.\ P\ x \longrightarrow Q) \longrightarrow Q \\
\text{False} & \equiv & \forall P.\ P \\
\neg P & \equiv & P \longrightarrow \text{False} \\
P \wedge Q & \equiv & \forall R.\ (P \longrightarrow Q \longrightarrow R) \longrightarrow R \\
P \vee Q & \equiv & \forall R.\ (P \longrightarrow R) \longrightarrow (Q \longrightarrow R) \longrightarrow R \\
\text{If } P\ x\ y & \equiv & \text{SOME } z.\ (P = \text{True} \longrightarrow z = x) \wedge (P = \text{False} \longrightarrow z = y) \\
\text{inj } f & \equiv & \forall x\ y.\ f\ x = f\ y \longrightarrow x = y \\
\text{surj } f & \equiv & \forall y.\ \exists x.\ y = f\ x
\end{array}
$$

The Axioms of HOL

$$\frac{}{t = t}\ \text{refl} \qquad \frac{s = t \quad P\ s}{P\ t}\ \text{subst} \qquad \frac{\bigwedge x.\ f\ x = g\ x}{(\lambda x.\ f\ x) = (\lambda x.\ g\ x)}\ \text{ext}$$

$$\frac{P \Longrightarrow Q}{P \longrightarrow Q}\ \text{impl} \qquad \frac{P \longrightarrow Q \quad P}{Q}\ \text{mp}$$

$$\frac{}{(P \longrightarrow Q) \longrightarrow (Q \longrightarrow P) \longrightarrow (P = Q)}\ \text{iff}$$

$$\frac{}{P = \text{True} \vee P = \text{False}}\ \text{True\_or\_False}$$

$$\frac{P\ ?x}{P\ (\text{SOME}\ x.\ P\ x)}\ \text{somel}$$

$$\frac{}{\exists f :: ind \Rightarrow ind.\ \text{inj}\ f \wedge \neg \text{surj}\ f}\ \text{infty}$$

That's it.

- 3 basic constants
- 3 basic types
- 9 axioms

With this you can define and derive all the rest.

Isabelle knows 2 more axioms:

$$\frac{x = y}{x \equiv y}\ \text{eq\_reflection} \qquad \frac{}{(\text{THE } x.\ x = a) = a}\ \text{the\_eq\_trivial}$$

Demo:
The Definitions in Isabelle

Deriving Proof Rules

In the following, we will

- look at the definitions in more detail
- derive the traditional proof rules from the axioms in Isabelle

Convenient for deriving rules: **named assumptions in lemmas**

lemma [*name* :]
assumes [$name_1$:] “$< prop >_1$”
assumes [$name_2$:] “$< prop >_2$”
⋮
shows “$< prop >$” $< proof >$

proves: $[\![< prop >_1;\ < prop >_2;\ \ldots]\!] \Longrightarrow < prop >$

True

consts True :: *bool*
$\text{True} \equiv (\lambda x :: bool.\ x) = (\lambda x.\ x)$

Intuition:
right hand side is always true

Proof Rules:

$$\frac{}{\text{True}}\ \text{TrueI}$$

Proof:

$$\frac{\dfrac{}{(\lambda x :: bool.\ x) = (\lambda x.\ x)}\ \text{refl}}{\text{True}}\ \text{unfold True\_def}$$

Demo

Universal Quantifier

consts ALL :: $(\alpha \Rightarrow bool) \Rightarrow bool$
ALL $P \equiv P = (\lambda x.\ \text{True})$

Intuition:

- ALL P is Higher Order Abstract Syntax for $\forall x.\ P\ x$.
- P is a function that takes an x and yields a truth value.
- ALL P should be true iff P yields true for all x, i.e. if it is equivalent to the function $\lambda x.$ True.

Proof Rules:

$$\frac{\bigwedge x.\ P\ x}{\forall x.\ P\ x}\ \text{allI} \qquad \frac{\forall x.\ P\ x \quad P\ ?x \Longrightarrow R}{R}\ \text{allE}$$

Proof: Isabelle Demo

False

consts False :: *bool*
False $\equiv$ $\forall P.P$

Intuition:
Everything can be derived from *False*.

Proof Rules:

$$\frac{\text{False}}{P}\ \text{FalseE} \qquad \frac{}{\text{True} \neq \text{False}}$$

Proof: Isabelle Demo

Negation

consts Not :: $bool \Rightarrow bool$ $(\neg \_)$
$\neg P \equiv P \longrightarrow \text{False}$

Intuition:
Try $P = True$ and $P = False$ and the traditional truth table for $\longrightarrow$.

Proof Rules:

$$\frac{A \Longrightarrow False}{\neg A}\ \text{notI} \qquad \frac{\neg A \quad A}{P}\ \text{notE}$$

Proof: Isabelle Demo

Existential Quantifier

consts EX :: $(\alpha \Rightarrow bool) \Rightarrow bool$
EX $P \equiv \forall Q.\ (\forall x.\ P\ x \longrightarrow Q) \longrightarrow Q$

Intuition:

- EX P is HOAS for $\exists x.\ P\ x$. (like $\forall$)
- Right hand side is characterization of $\exists$ with $\forall$ and $\longrightarrow$
- Note that inner $\forall$ binds wide: $(\forall x.\ P\ x \longrightarrow Q)$
- Remember lemma from last time:
  $(\forall x.\ P\ x \longrightarrow Q) = ((\exists x.\ P\ x) \longrightarrow Q)$

Proof Rules:

$$\frac{P\ ?x}{\exists x.\ P\ x}\ \text{exI} \qquad \frac{\exists x.\ P\ x \quad \bigwedge x.\ P\ x \Longrightarrow R}{R}\ \text{exE}$$

Proof: Isabelle Demo

Conjunction

consts And :: $bool \Rightarrow bool \Rightarrow bool$ $(\_ \wedge \_)$
$P \wedge Q \equiv \forall R.\ (P \longrightarrow Q \longrightarrow R) \longrightarrow R$

Intuition:

- Mirrors proof rules for $\wedge$
- Try truth table for P, Q, and R

Proof Rules:

$$\frac{A \quad B}{A \wedge B}\ \text{conjI} \qquad \frac{A \wedge B \quad [\![A; B]\!] \Longrightarrow C}{C}\ \text{conjE}$$

Proof: Isabelle Demo

Disjunction

consts Or :: $bool \Rightarrow bool \Rightarrow bool$ $(\_ \vee \_)$
$P \vee Q \equiv \forall R.\ (P \longrightarrow R) \longrightarrow (Q \longrightarrow R) \longrightarrow R$

Intuition:

- Mirrors proof rules for $\vee$ (case distinction)
- Try truth table for P, Q, and R

Proof Rules:

$$\frac{A}{A \vee B}\ \frac{B}{A \vee B}\ \text{disjI1/2} \qquad \frac{A \vee B \quad A \Longrightarrow C \quad B \Longrightarrow C}{C}\ \text{disjE}$$

Proof: Isabelle Demo

If-Then-Else

consts If :: $bool \Rightarrow \alpha \Rightarrow \alpha \Rightarrow \alpha$ (if_ then _ else _)
If $P\ x\ y \equiv$ SOME z. $(P = \text{True} \longrightarrow z = x) \wedge (P = \text{False} \longrightarrow z = y)$

Intuition:

- for $P = \text{True}$, right hand side collapses to SOME z. $z = x$
- for $P = \text{False}$, right hand side collapses to SOME z. $z = y$

Proof Rules:

$$\frac{}{\text{if True then } s \text{ else } t = s}\ \text{ifTrue} \qquad \frac{}{\text{if False then } s \text{ else } t = t}\ \text{ifFalse}$$

Proof: Isabelle Demo

That was HOL

We have learned ...

- Defining HOL
- Higher Order Abstract Syntax
- Deriving proof rules

Section 9

Term Rewriting

The Problem

Given a set of equations

$$
\begin{aligned}
l_1 &= r_1 \\
l_2 &= r_2 \\
&\vdots \\
l_n &= r_n
\end{aligned}
$$

does equation $l = r$ hold?

Applications in:

- Mathematics (algebra, group theory, etc)
- Functional Programming (model of execution)
- Theorem Proving (dealing with equations, simplifying statements)

Term Rewriting: The Idea

use equations as reduction rules

$$\begin{array}{c} l_1 \longrightarrow r_1 \\ l_2 \longrightarrow r_2 \\ \vdots \\ l_n \longrightarrow r_n \end{array}$$

decide $l = r$ by deciding $l \overset{*}{\longleftrightarrow} r$

Arrow Cheat Sheet

$\xrightarrow{0}$	$=$	$\{(x,y)\|x=y\}$	identity
$\xrightarrow{n+1}$	$=$	$\xrightarrow{n} \circ \longrightarrow$	n+1 fold composition
$\xrightarrow{+}$	$=$	$\bigcup_{i>0} \xrightarrow{i}$	transitive closure
$\xrightarrow{*}$	$=$	$\xrightarrow{+} \cup \xrightarrow{0}$	reflexive transitive closure
$\xrightarrow{=}$	$=$	$\longrightarrow \cup \xrightarrow{0}$	reflexive closure
$\xrightarrow{-1}$	$=$	$\{(y,x)\|x \longrightarrow y\}$	inverse
$\longleftarrow$	$=$	$\xrightarrow{-1}$	inverse
$\longleftrightarrow$	$=$	$\longleftarrow \cup \longrightarrow$	symmetric closure
$\xleftrightarrow{+}$	$=$	$\bigcup_{i>0} \xleftrightarrow{i}$	transitive symmetric closure
$\xleftrightarrow{*}$	$=$	$\xleftrightarrow{+} \cup \xleftrightarrow{0}$	reflexive transitive symmetric closure

How to Decide $l \overset{*}{\longleftrightarrow} r$

Same idea as for β: look for n such that $l \overset{*}{\longrightarrow} n$ and $r \overset{*}{\longrightarrow} n$

Does this always work?

If $l \overset{*}{\longrightarrow} n$ and $r \overset{*}{\longrightarrow} n$ then $l \overset{*}{\longleftrightarrow} r$. Ok.
If $l \overset{*}{\longleftrightarrow} r$, will there always be a suitable n? **No!**

Example:

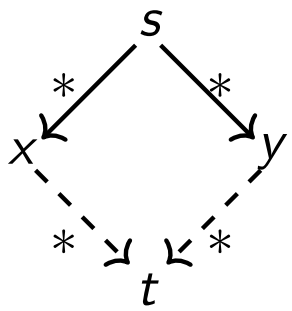
Rules: $f\ x \longrightarrow a, \quad g\ x \longrightarrow b, \quad f\ (g\ x) \longrightarrow b$
$f\ x \overset{*}{\longleftrightarrow} g\ x$ because $f\ x \longrightarrow a \longleftarrow f\ (g\ x) \longrightarrow b \longleftarrow g\ x$
But: $f\ x \longrightarrow a$ and $g\ x \longrightarrow b$ and a, b in normal form

Works only for systems with **Church-Rosser** property:

$$l \overset{*}{\longleftrightarrow} r \Longrightarrow \exists n.\ l \overset{*}{\longrightarrow} n \wedge r \overset{*}{\longrightarrow} n$$

Fact: $\longrightarrow$ is Church-Rosser iff it is confluent.

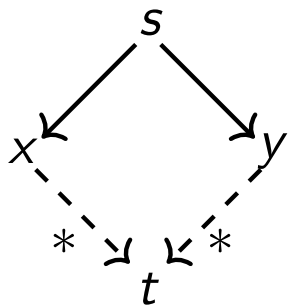
Confluence



Problem:
is a given set of reduction rules confluent?

undecidable

Local Confluence



Fact: local confluence and termination $\Longrightarrow$ confluence

Termination

- $\longrightarrow$ is **terminating** if there are no infinite reduction chains
- $\longrightarrow$ is **normalizing** if each element has a normal form
- $\longrightarrow$ is **convergent** if it is terminating and confluent

Example:

- $\longrightarrow_\beta$ in λ is not terminating, but confluent
- $\longrightarrow_\beta$ in $\lambda^\rightarrow$ is terminating and confluent, i.e. convergent

Problem: is a given set of reduction rules terminating?

undecidable

When is $\longrightarrow$ Terminating?

Basic idea: when each rule application makes terms simpler in some way.

More formally: $\longrightarrow$ is terminating when there is a well founded order $<$ on terms for which $s < t$ whenever $t \longrightarrow s$
(well founded = no infinite decreasing chains $a_1 > a_2 > \ldots$)

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

This system always terminates. Reduction order:

$s <_r t$ iff $\mathit{size}(s) < \mathit{size}(t)$ with
$\mathit{size}(s) =$ number of function symbols in s

1. Both rules always decrease size by 1 when applied to any term t
2. $<_r$ is well founded, because $<$ is well founded on $\mathbb{N}$

Termination in Practice

In practice: often easier to consider just the rewrite rules by themselves, rather than their application to an arbitrary term t.

Show for each rule $l_i = r_i$, that $r_i < l_i$.

Example:

$g\ x < f\ (g\ x)$ and $f\ x < g\ (f\ x)$

Requires

u to become smaller whenever any subterm of u is made smaller.

Formally:

Requires $<$ to be **monotonic** with respect to the structure of terms:

$$s < t \longrightarrow u[s] < u[t].$$

True for most orders that don't treat certain parts of terms as special cases.

Example Termination Proof

Problem: Rewrite formulae containing $\neg$, $\wedge$, $\vee$ and $\longrightarrow$, so that they don't contain any implications and $\neg$ is applied only to variables and constants.

Rewrite Rules:

- Remove implications:

  imp: $(A \longrightarrow B) = (\neg A \vee B)$

- Push $\neg$s down past other operators:

  notnot: $(\neg\neg P) = P$

  notand: $(\neg(A \wedge B)) = (\neg A \vee \neg B)$

  notor: $(\neg(A \vee B)) = (\neg A \wedge \neg B)$

We show that the rewrite system defined by these rules is terminating.

Order on Terms

Each time one of our rules is applied, either:

- an implication is removed, or
- something that is not a $\neg$ is hoisted upwards in the term.

This suggests a 2-part order, $<_r$: $s <_r t$ iff:

- $\mathsf{num\_imps}\ s < \mathsf{num\_imps}\ t$, or
- $\mathsf{num\_imps}\ s = \mathsf{num\_imps}\ t \wedge \mathsf{osize}\ s < \mathsf{osize}\ t$.

Let:

- $s <_i t \equiv \mathsf{num\_imps}\ s < \mathsf{num\_imps}\ t$ and
- $s <_n t \equiv \mathsf{osize}\ s < \mathsf{osize}\ t$

Then $<_i$ and $<_n$ are both well-founded orders (since both return nats). $<_r$ is the lexicographic order over $<_i$ and $<_n$. $<_r$ is well-founded since $<_i$ and $<_n$ are both well-founded.

Order Decreasing

imp clearly decreases num_imps.
osize adds up all non-$\neg$ operators and variables/constants, weights each one according to its depth within the term.

$$
\begin{array}{ll}
\text{osize}'\ c & x = 2^x \\
\text{osize}'\ (\neg P) & x = \text{osize}'\ P\ (x+1) \\
\text{osize}'\ (P \wedge Q) & x = 2^x + (\text{osize}'\ P\ (x+1)) + (\text{osize}'\ Q\ (x+1)) \\
\text{osize}'\ (P \vee Q) & x = 2^x + (\text{osize}'\ P\ (x+1)) + (\text{osize}'\ Q\ (x+1)) \\
\text{osize}'\ (P \longrightarrow Q) & x = 2^x + (\text{osize}'\ P\ (x+1)) + (\text{osize}'\ Q\ (x+1)) \\
\text{osize}\ \ P & = \text{osize}'\ P\ 0
\end{array}
$$

The other rules decrease the depth of the things osize counts, so decrease osize.

Term Rewriting in Isabelle

Term rewriting engine in Isabelle is called **Simplifier**

apply simp

- uses simplification rules
- (almost) blindly from left to right
- until no rule is applicable.

termination:	not guaranteed (may loop)
confluence:	not guaranteed (result may depend on which rule is used first)

Control

- Equations turned into simplification rules with **[simp]** attribute
- Adding/deleting equations locally:

  apply (simp add: <rules>) and **apply** (simp del: <rules>)
- Using only the specified set of equations:

  apply (simp only: <rules>)

Demo

Exercises

- Show, via a pen-and-paper proof, that the osize function is monotonic with respect to the structure of terms from that example.

Applying a Rewrite Rule

- $l \longrightarrow r$ applicable to term $t[s]$
  if there is substitution σ such that $\sigma\ l = s$
- Result: $t[\sigma\ r]$
- Equationally: $t[s] = t[\sigma\ r]$

Example:

Rule: $0 + n \longrightarrow n$

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

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

Result: $a + (b + c)$

Conditional Term Rewriting

Rewrite rules can be conditional:

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

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

- $\sigma\ l = s$ and
- $\sigma\ P_1, \ldots, \sigma\ P_n$ are provable by rewriting.

Rewriting with Assumptions

Isabelle uses assumptions in rewriting.

Can lead to non-termination.

Example:

lemma "$f\ x = g\ x \wedge g\ x = f\ x \Longrightarrow f\ x = 2$"

simp	**use and simplify** assumptions
(simp (no_asm))	**ignore** assumptions
(simp (no_asm_use))	**simplify**, but do **not use** assumptions
(simp (no_asm_simp))	**use**, but do **not simplify** assumptions

Preprocessing

Preprocessing (recursive) for maximal simplification power:

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

Example:

$$(p \longrightarrow q \wedge \neg r) \wedge s$$

$$\mapsto$$

$$p \Longrightarrow q = \mathit{True} \qquad p \Longrightarrow r = \mathit{False} \qquad s = \mathit{True}$$

Demo

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

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

Manually: **apply** (simp split: nat.split)

Similar for any data type t: **t.split**

Congruence Rules

congruence rules are about using context

Example: in $P \longrightarrow Q$ we could use P to simplify terms in Q

For $\Longrightarrow$ hardwired (assumptions used in rewriting)

For other operators expressed with conditional rewriting.

Example: $[\![P = P'; P' \Longrightarrow Q = Q']\!] \Longrightarrow (P \longrightarrow Q) = (P' \longrightarrow Q')$

Read: to simplify $P \longrightarrow Q$

- first simplify P to P'
- then simplify Q to Q' using P' as assumption
- the result is $P' \longrightarrow Q'$

More Congruence

Sometimes useful, but not used automatically (slowdown):

conj_cong: $[\![P = P'; P' \Longrightarrow Q = Q']\!] \Longrightarrow (P \wedge Q) = (P' \wedge Q')$

Context for if-then-else:

if_cong: $[\![b = c; c \Longrightarrow x = u; \neg c \Longrightarrow y = v]\!] \Longrightarrow$
$(\text{if } b \text{ then } x \text{ else } y) = (\text{if } c \text{ then } u \text{ else } v)$

Prevent rewriting inside then-else (default):

if_weak_cong: $b = c \Longrightarrow (\text{if } b \text{ then } x \text{ else } y) = (\text{if } c \text{ then } x \text{ else } y)$

- declare own congruence rules with **[cong]** attribute
- delete with **[cong del]**
- use locally with e.g. **apply** (simp cong: <rule>)

Ordered rewriting

Problem: $x + y \longrightarrow y + x$ does not terminate

Solution: use permutative rules only if 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 **mult_ac** sort any product $(*)$

Example: **apply** (simp add: add_ac) yields
$(b + c) + a \rightsquigarrow \cdots \rightsquigarrow a + (b + c)$

AC Rules

Example for associative-commutative rules:

Associative: $(x \odot y) \odot z = x \odot (y \odot z)$

Commutative: $x \odot y = y \odot x$

These 2 rules alone get stuck too early (not confluent).

Example: $(z \odot x) \odot (y \odot v)$

We want: $(z \odot x) \odot (y \odot v) = v \odot (x \odot (y \odot z))$

We get: $(z \odot x) \odot (y \odot v) = v \odot (y \odot (x \odot z))$

We need: **AC rule** $x \odot (y \odot z) = y \odot (x \odot z)$

If these 3 rules are present for an AC operator
Isabelle will order terms correctly

Demo

Back to Confluence

Remember: confluence in general is undecidable.
But: confluence for terminating systems is decidable!
Problem: overlapping lhs of rules.

Definition:

Let $l_1 \longrightarrow r_1$ and $l_2 \longrightarrow r_2$ be two rules with disjoint variables.
They form a **critical pair** if a non-variable subterm of l_1 unifies with l_2.

Example:

Rules: (1) $f\ x \longrightarrow a$ (2) $g\ y \longrightarrow b$ (3) $f\ (g\ z) \longrightarrow b$

Critical pairs:

(1)+(3)	$\{x \mapsto g\ z\}$	$a \xleftarrow{(1)} f\ (g\ z) \xrightarrow{(3)} b$
(3)+(2)	$\{z \mapsto y\}$	$b \xleftarrow{(3)} f\ (g\ y) \xrightarrow{(2)} f\ b$

Completion

$$(1)\ f\ x \longrightarrow a \quad (2)\ g\ y \longrightarrow b \quad (3)\ f\ (g\ z) \longrightarrow b$$

is not confluent

But it can be made confluent by adding rules!

How: join all critical pairs

Example:

(1)+(3) $\quad \{x \mapsto g\ z\} \quad a \xleftarrow{(1)} f\ (g\ z) \xrightarrow{(3)} b$

shows that $a = b$ (because $a \overset{*}{\longleftrightarrow} b$), so we add $a \longrightarrow b$ as a rule

This is the main idea of the Knuth-Bendix completion algorithm.

Orthogonal Rewriting Systems

Definitions:
A **rule** $l \longrightarrow r$ is **left-linear** if no variable occurs twice in l.
A **rewrite system** is **left-linear** if all rules are.

A system is **orthogonal** if it is left-linear and has no critical pairs.

Orthogonal rewrite systems are confluent

Application: functional programming languages

We have learned ...

- Conditional term rewriting
- Congruence rules
- AC rules
- More on confluence

Specification Techniques

Section 10

Sets, Types & Rule Induction

Sets in Isabelle

Type **'a set**: sets over type 'a

- $\{\}, \quad \{e_1, \ldots, e_n\}, \quad \{x.\ P\ x\}$
- $e \in A, \quad A \subseteq B$
- $A \cup B, \quad A \cap B, \quad A - B, \quad -A$
- $\bigcup x \in A.\ B\ x, \quad \bigcap x \in A.\ B\ x, \quad \bigcap A, \quad \bigcup A$
- $\{i..j\}$
- insert :: $\alpha \Rightarrow \alpha$ set $\Rightarrow \alpha$ set
- $f`A \equiv \{y.\ \exists x \in A.\ y = f\ x\}$
- ...

Proofs about Sets

Natural deduction proofs:

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

Bounded Quantifiers

- $\forall x \in A.\ P\ x \equiv \forall x.\ x \in A \longrightarrow P\ x$
- $\exists x \in A.\ P\ x \equiv \exists x.\ x \in A \wedge P\ x$
- 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: $[\![\exists x \in A.\ P\ x; \bigwedge x.\ [\![x \in A; P\ x]\!] \Longrightarrow Q]\!] \Longrightarrow Q$

Demo: Sets

The Three Basic Ways of Introducing Theorems

- **Axioms**:

  Example: **axiomatization where** refl: "$t = t$"

  Do not use. Evil. Can make your logic inconsistent.

- **Definitions:**

  Example: **definition** inj **where**
  "inj $f \equiv \forall x\ y.\ f\ x = f\ y \longrightarrow x = y$"
  Introduces a new lemma called inj_def.

- **Proofs:**

  Example: **lemma** "inj $(\lambda x.\ x + 1)$"

  The harder, but safe choice.

The Three Basic Ways of Introducing Types

- **typedecl**: by name only

  Example: **typedecl** names
  Introduces new type *names* without any further assumptions

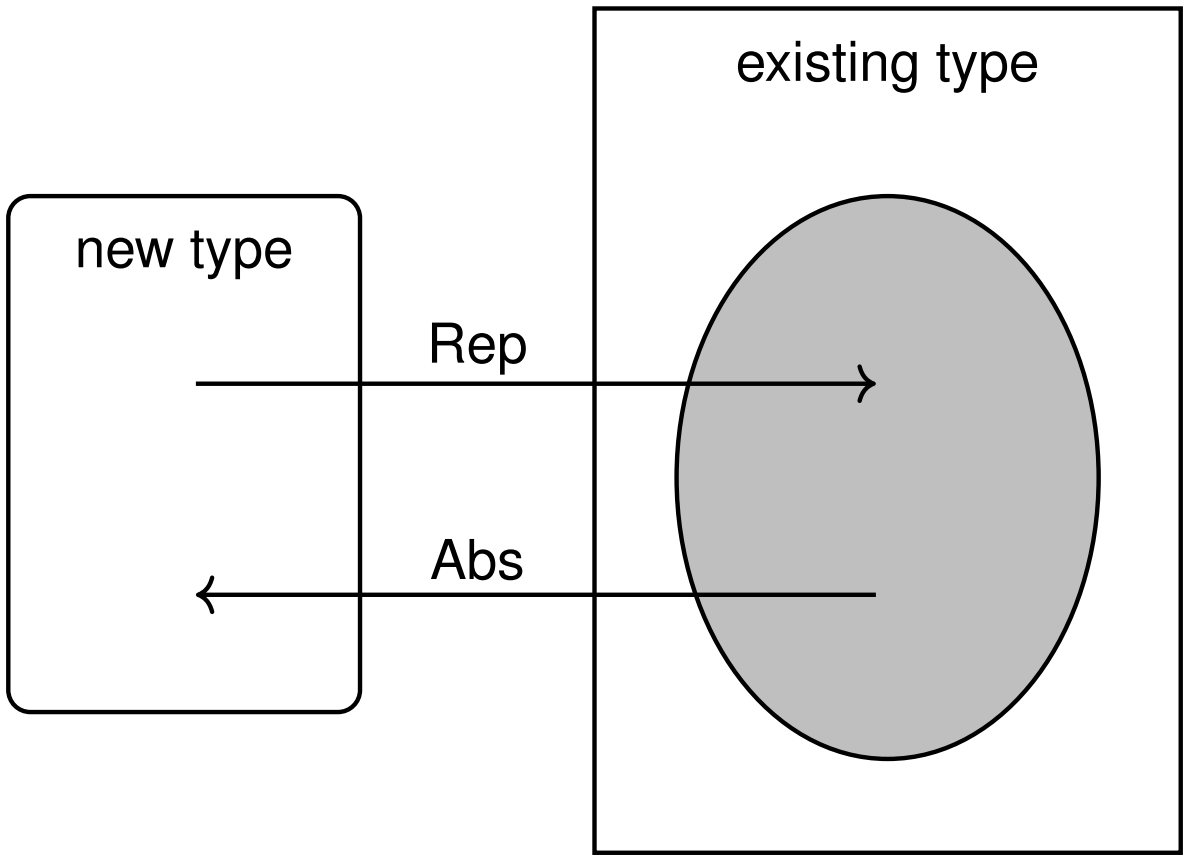
- **type_synonym**: by abbreviation

  Example: **type_synonym** α rel = "$\alpha \Rightarrow \alpha \Rightarrow$ *bool*"
  Introduces abbreviation *rel* for existing type $\alpha \Rightarrow \alpha \Rightarrow$ *bool*
  Type abbreviations are immediately expanded internally

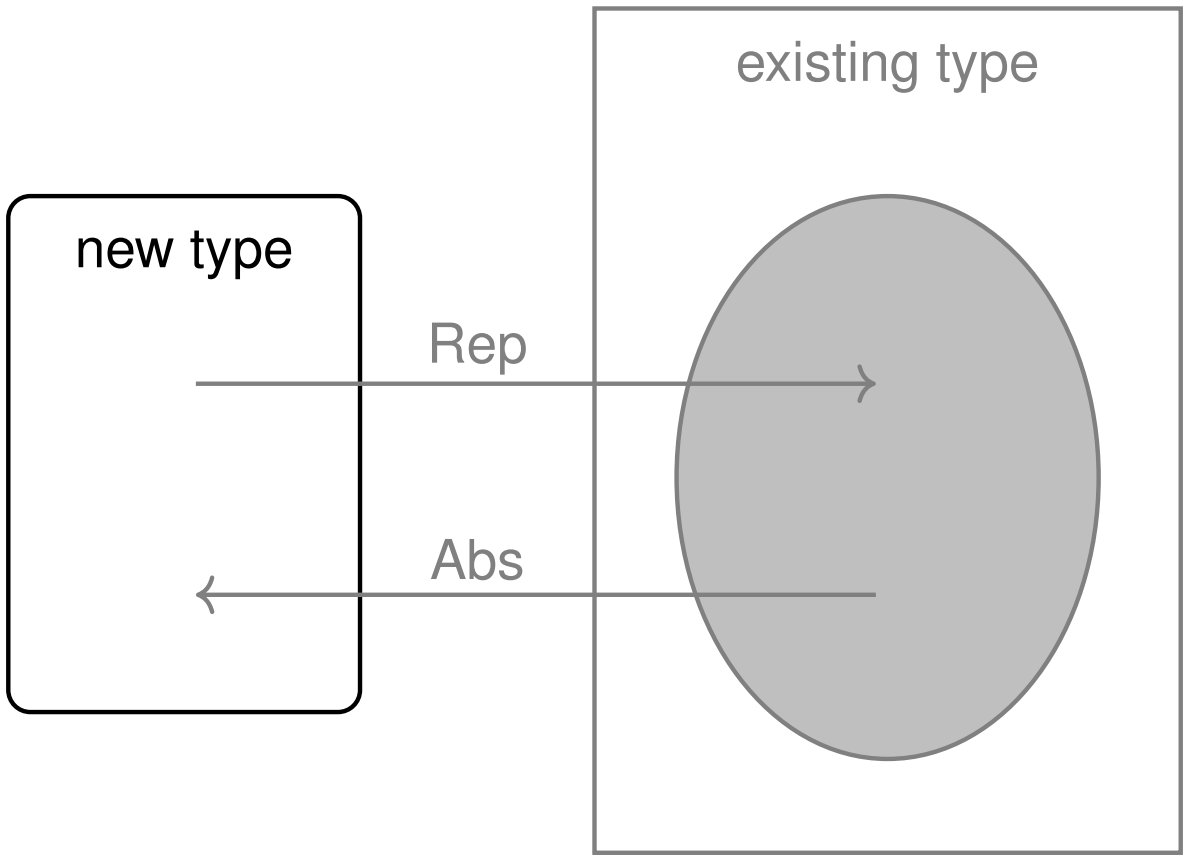
- **typedef**: by definiton as a set

  Example: **typedef** new_type = "{some set}" <proof>
  Introduces a new type as a subset of an existing type.
  The proof shows that the set on the rhs in non-empty.

How typedef works



How typedef works



Example: Pairs

$$(\alpha, \beta)\ \text{Prod}$$

1. Pick existing type: $\alpha \Rightarrow \beta \Rightarrow \text{bool}$
2. Identify subset:
   $(\alpha, \beta)\ \text{Prod} = \{f.\ \exists a\ b.\ f = \lambda(x :: \alpha)\ (y :: \beta).\ x = a \wedge y = b\}$
3. We get from Isabelle:
   - functions Abs_Prod, Rep_Prod
   - both injective
   - Abs_Prod (Rep_Prod x) $= x$
4. We now can:
   - define constants Pair, fst, snd in terms of Abs_Prod and Rep_Prod
   - derive all characteristic theorems
   - forget about Rep/Abs, use characteristic theorems instead

Demo: Introducing new Types

Inductive Definitions

Example

$$\frac{}{\langle \text{skip}, \sigma \rangle \longrightarrow \sigma} \qquad \frac{[\![e]\!]\sigma = v}{\langle \text{x} := \text{e}, \sigma \rangle \longrightarrow \sigma[x \mapsto v]}$$

$$\frac{\langle c_1, \sigma \rangle \longrightarrow \sigma' \quad \langle c_2, \sigma' \rangle \longrightarrow \sigma''}{\langle c_1; c_2, \sigma \rangle \longrightarrow \sigma''}$$

$$\frac{[\![b]\!]\sigma = \text{False}}{\langle \text{while } b \text{ do } c, \sigma \rangle \longrightarrow \sigma}$$

$$\frac{[\![b]\!]\sigma = \text{True} \quad \langle c, \sigma \rangle \longrightarrow \sigma' \quad \langle \text{while } b \text{ do } c, \sigma' \rangle \longrightarrow \sigma''}{\langle \text{while } b \text{ do } c, \sigma \rangle \longrightarrow \sigma''}$$

What does this mean?

- $\langle c, \sigma \rangle \longrightarrow \sigma'$ fancy syntax for a relation $(c, \sigma, \sigma') \in E$
- relations are sets: $E :: (\mathsf{com} \times \mathsf{state} \times \mathsf{state})\ \mathsf{set}$
- the rules define a set inductively

But which set?

Simpler Example

$$\frac{}{0 \in N} \qquad \frac{n \in N}{n+1 \in N}$$

- N is the set of natural numbers $\mathbb{N}$
- But why not the set of real numbers? $0 \in \mathbb{R}$, $n \in \mathbb{R} \Longrightarrow n+1 \in \mathbb{R}$
- $\mathbb{N}$ is the smallest set that is consistent with the rules.

Why the smallest set?

- Objective: no junk. Only what must be in X shall be in X.
- Gives rise to a nice proof principle (rule induction)
- Alternative (greatest set) occasionally also useful: coinduction

Rule Induction

$$\frac{}{0 \in N} \qquad \frac{n \in N}{n+1 \in N}$$

induces induction principle

$$[\![P\ 0;\ \bigwedge n.\ P\ n \Longrightarrow P\ (n+1)]\!] \Longrightarrow \forall x \in N.\ P\ x$$

Demo: Inductive Definitions

Formally

$$\text{Rules } \frac{a_1 \in X \quad \ldots \quad a_n \in X}{a \in X} \text{ with } a_1, \ldots, a_n, a \in A$$

$$\text{define set } X \subseteq A$$

Formally: set of rules $R \subseteq A \text{ set} \times A$ (R, X possibly infinite)

Applying rules R to a set B: $\hat{R}\ B \equiv \{x.\ \exists H.\ (H, x) \in R \wedge H \subseteq B\}$

Example:

$$\begin{array}{lcl} R & \equiv & \{(\{\}, 0)\} \cup \{(\{n\}, n+1).\ n \in \mathbf{R}\} \\ \hat{R}\ \{3, 6, 10\} & = & \{0, 4, 7, 11\} \end{array}$$

The Set

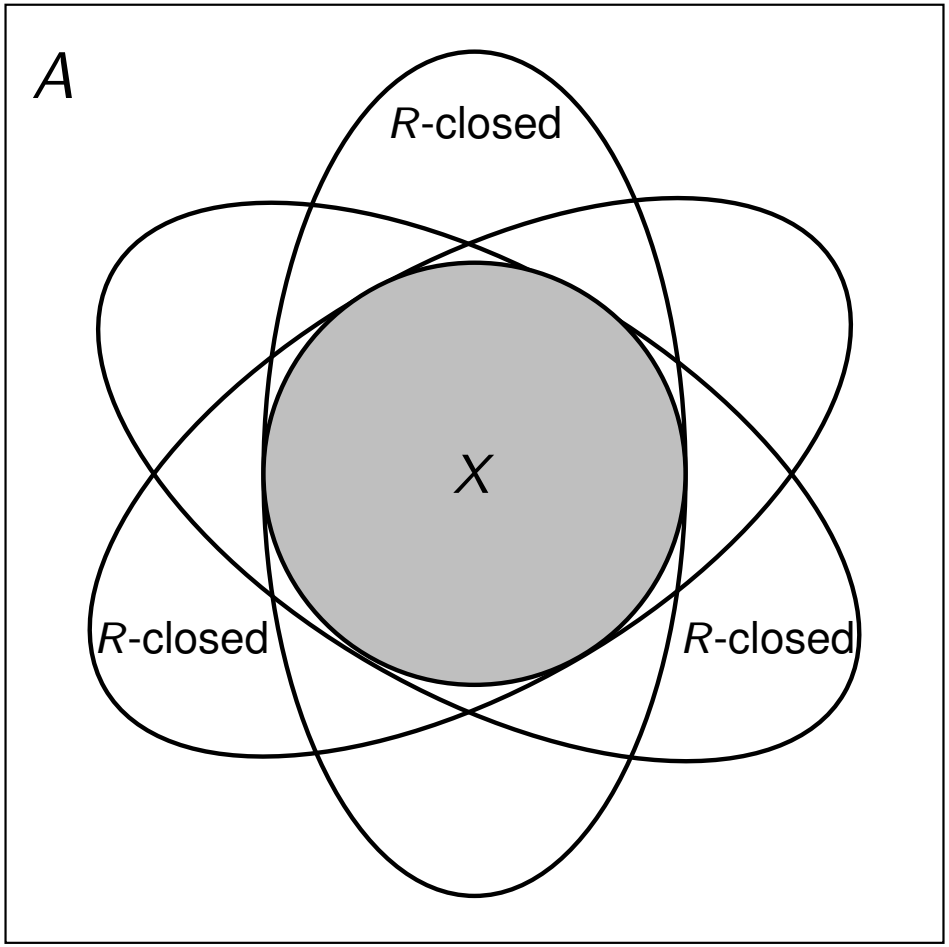
Definition: B is R-closed iff $\hat{R}\ B \subseteq B$

Definition: X is the least R-closed subset of A

This does always exist:

Fact: $X = \bigcap\{B \subseteq A.\ B\ R\text{-closed}\}$

Generation from Above



Rule Induction

$$\frac{}{0 \in N} \qquad \frac{n \in N}{n+1 \in N}$$

induces induction principle

$$[\![P\ 0;\ \bigwedge n.\ P\ n \Longrightarrow P\ (n+1)]\!] \Longrightarrow \forall x \in N.\ P\ x$$

In general:

$$\frac{\forall(\{a_1, \ldots a_n\}, a) \in R.\ P\ a_1 \wedge \ldots \wedge P\ a_n \Longrightarrow P\ a}{\forall x \in X.\ P\ x}$$

Why does this work?

$$\frac{\forall(\{a_1, \ldots a_n\}, a) \in R.\ P\ a_1 \wedge \ldots \wedge P\ a_n \Longrightarrow P\ a}{\forall x \in X.\ P\ x}$$

$\forall(\{a_1, \ldots a_n\}, a) \in R.\ P\ a_1 \wedge \ldots \wedge P\ a_n \Longrightarrow P\ a$

says

$\{x.\ P\ x\}$ is R-closed

but:	X is the least R-closed set
hence:	$X \subseteq \{x.\ P\ x\}$
which means:	$\forall x \in X.\ P\ x$

qed

Rules with side conditions

$$\frac{a_1 \in X \quad \ldots \quad a_n \in X \qquad C_1 \quad \ldots \quad C_m}{a \in X}$$

induction scheme:

$$(\forall(\{a_1, \ldots a_n\}, a) \in R.\ P\ a_1 \wedge \ldots \wedge P\ a_n \wedge C_1 \wedge \ldots \wedge C_m \wedge \{a_1, \ldots, a_n\} \subseteq X \Longrightarrow P\ a)$$

$$\Longrightarrow$$

$$\forall x \in X.\ P\ x$$

X as Fixpoint

How to compute X?

$X = \bigcap\{B \subseteq A.\ B\ R - \text{closed}\}$ hard to work with.

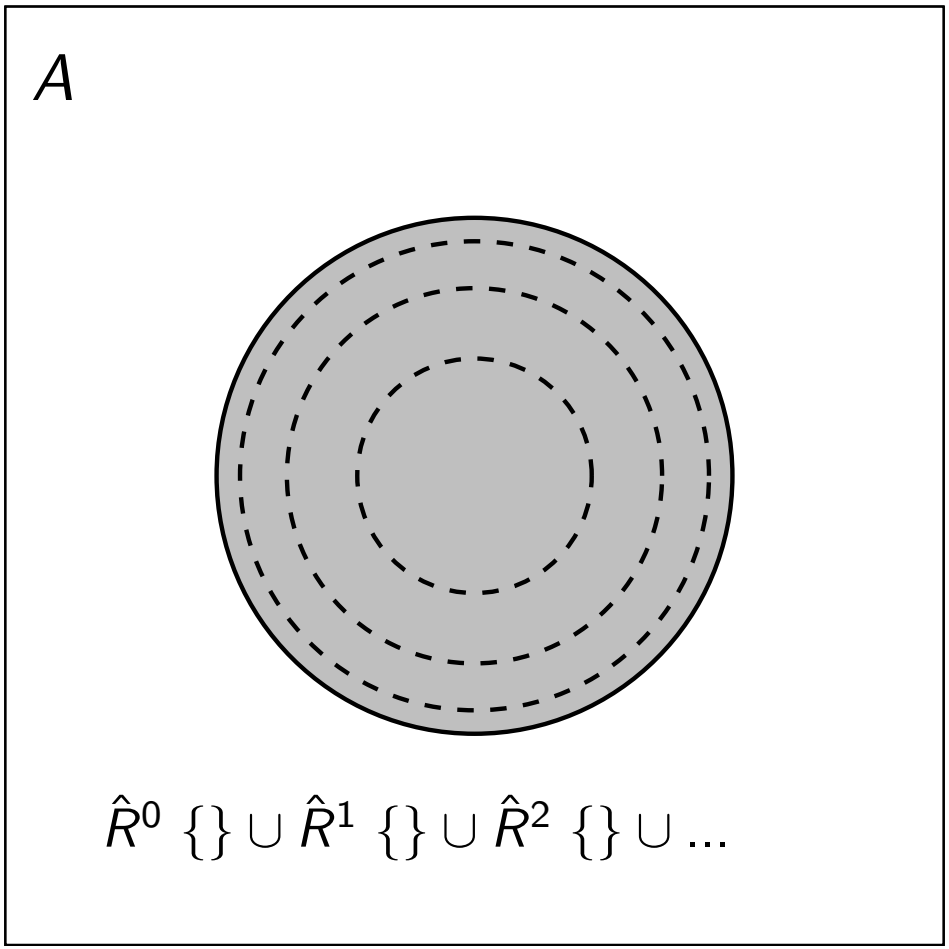
Instead: view X as least fixpoint, X least set with $\hat{R}\ X = X$.

Fixpoints can be approximated by iteration:

$$X_0 = \hat{R}^0\ \{\} = \{\}$$
$$X_1 = \hat{R}^1\ \{\} = \text{rules without hypotheses}$$
$$\vdots$$
$$X_n = \hat{R}^n\ \{\}$$

$$X_\omega = \bigcup_{n \in \mathbf{N}}(\hat{R}^n\ \{\}) = X$$

Generation from Below



Does this always work?

Knaster-Tarski Fixpoint Theorem:
Let $(A, \leq)$ be a complete lattice, and $f :: A \Rightarrow A$ a monotone function. Then the fixpoints of f again form a complete lattice.

Lattice:
Finite subsets have a greatest lower bound (meet) and least upper bound (join).

Complete Lattice:
All subsets have a greatest lower bound and least upper bound.

Implications:

- least and greatest fixpoints exist (complete lattice always non-empty).
- can be reached by (possibly infinite) iteration. (Why?)

Exercise

Formalise this lecture in Isabelle:

- Define **closed** $f\ A :: (\alpha\ \text{set} \Rightarrow \alpha\ \text{set}) \Rightarrow \alpha\ \text{set} \Rightarrow \text{bool}$
- Show $\text{closed}\ f\ A \wedge \text{closed}\ f\ B \Longrightarrow \text{closed}\ f\ (A \cap B)$ if f is monotone (**mono** is predefined)
- Define **lfpt** f as the intersection of all f-closed sets
- Show that lfpt f is a fixpoint of f if f is monotone
- Show that lfpt f is the least fixpoint of f
- Declare a constant $R :: (\alpha\ \text{set} \times \alpha)\ \text{set}$
- Define $\hat{R} :: \alpha\ \text{set} \Rightarrow \alpha\ \text{set}$ in terms of R
- Show soundness of rule induction using R and lfpt $\hat{R}$

We have learned ...

- Formal background of inductive definitions
- Definition by intersection
- Computation by iteration
- Formalisation in Isabelle

Section 11

Datatypes

Datatypes

Example:

datatype 'a list = Nil | Cons 'a "'a list"

Properties:

- Constructors:

$$\begin{array}{lll} \text{Nil} & :: & \text{'a list} \\ \text{Cons} & :: & \text{'a} \Rightarrow \text{'a list} \Rightarrow \text{'a list} \end{array}$$

- Distinctness: $\text{Nil} \neq \text{Cons x xs}$
- Injectivity: $(\text{Cons x xs} = \text{Cons y ys}) = (\text{x} = \text{y} \wedge \text{xs} = \text{ys})$

More Examples

Enumeration:

datatype answer = Yes | No | Maybe

Polymorphic:

datatype 'a option = None | Some 'a
datatype ('a,'b,'c) triple = Triple 'a 'b 'c

Recursion:

datatype 'a list = Nil | Cons 'a "'a list"
datatype 'a tree = Tip | Node 'a "'a tree" "a tree"

Mutual Recursion:

datatype even = EvenZero | EvenSucc odd
and odd = OddSucc even

Nested

Nested recursion:

datatype 'a tree = Tip | Node 'a "'a tree list"

datatype 'a tree = Tip | Node 'a "'a tree option" "'a tree option"

- Recursive call is under a type constructor.

The General Case

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

- Constructors: $C_i :: \tau_{i,1} \Rightarrow \ldots \Rightarrow \tau_{i,n_i} \Rightarrow (\alpha_1, \ldots, \alpha_n)\ \tau$
- 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 applied automatically

How is this Type Defined?

datatype 'a list = Nil | Cons 'a "'a list"

- internally reduced to a single constructor, using product and sum
- constructor defined as an inductive set (like typedef)
- recursion: least fixpoint

More detail: Tutorial on (Co-)datatypes Definitions at isabelle.in.tum.de

Datatype Limitations

Must be definable as a (non-empty) set.

- Infinitely branching ok.
- Mutually recursive ok.
- Strictly positive (right of function arrow) occurrence ok.

Not ok:

$$\begin{array}{lcl} \textbf{datatype}\ t & = & C\ (t \Rightarrow bool) \\ & | & D\ ((bool \Rightarrow t) \Rightarrow bool) \\ & | & E\ ((t \Rightarrow bool) \Rightarrow bool) \end{array}$$

Because: Cantor's theorem (α set is larger than α)

Datatype Limitations

Not ok (nested recursion):

datatype ('a, 'b) fun_copy = Fun "'a ⇒ 'b"

datatype 'a t = F "('a t, 'a) fun_copy"

- recursion in ('a1, ...,'an) t is only allowed on a subset of 'a1 ... 'an
- these arguments are called *live* arguments
- Mainly: in "'a ⇒ 'b", 'a is dead and 'b is live
- Thus: in ('a, 'b) fun_copy, 'a is dead and 'b is live
- type constructors must be registered as *BNFs** to have live arguments
- BNF defines well-behaved type constructors, ie where recursion is allowed
- datatypes automatically are BNFs (that's how they are constructed)
- can register other type constructors as BNFs — not covered here**

* BNF = Bounded Natural Functors.

** *Defining (Co)datatypes and Primitively (Co)recursive Functions in Isabelle/HOL*

Case

Every datatype introduces a **case** construct, e.g.

$$(\text{case } xs \text{ of } [] \Rightarrow \ldots \mid y \#ys \Rightarrow \ldots y \ldots ys \ldots)$$

In general: one case per constructor

- Nested patterns allowed: $x\#y\#zs$
- Dummy and default patterns with _
- Binds weakly, needs () in context

Cases

apply (case_tac t)

creates k subgoals

$$[\![t = C_i\ x_1 \ldots x_p; \ldots]\!] \Longrightarrow \ldots$$

one for each constructor C_i

Demo

Recursion

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 !

Primitive Recursion

primrec guarantees termination structurally

Example primrec:

```
primrec  app :: "'a list ⇒ 'a list ⇒ '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 $f :: \tau \Rightarrow \tau'$ can be defined by **primitive recursion**:

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

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

How does this Work?

primrec just fancy syntax for a **recursion operator**

Example:

rec_list :: "'a ⇒ ('b ⇒ 'b list ⇒ 'a ⇒ 'a) ⇒ 'b list ⇒ 'a"
rec_list f_1 f_2 Nil $=$ f_1
rec_list f_1 f_2 (Cons x xs) $=$ f_2 x xs (rec_list f_1 f_2 xs)

app $\equiv$ rec_list ($\lambda ys.\ ys$) ($\lambda x\ xs\ xs'.\ \lambda ys.$ Cons x (xs' ys))

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

rec_list

Defined: automatically, first inductively (set), then by epsilon

$$\frac{}{(\mathsf{Nil}, f_1) \in \mathsf{list\_rel}\ f_1\ f_2} \qquad \frac{(xs, xs') \in \mathsf{list\_rel}\ f_1\ f_2}{(\mathsf{Cons}\ x\ xs, f_2\ x\ xs\ xs') \in \mathsf{list\_rel}\ f_1\ f_2}$$

$$\mathsf{rec\_list}\ f_1\ f_2\ xs \equiv \mathsf{THE}\ y.\ (xs, y) \in \mathsf{list\_rel}\ f_1\ f_2$$

Automatic proof that set def indeed is total function
(the equations for rec_list are lemmas!)

Predefined Datatypes

nat is a datatype

datatype nat $= 0 \mid$ Suc nat

Functions on nat definable by primrec!

primrec

$$\begin{array}{lcl} f\ 0 & = & \ldots \\ f\ (\text{Suc}\ n) & = & \ldots\ f\ n\ \ldots \end{array}$$

Option

datatype 'a option = None | Some 'a

Important application:

'b ⇒ 'a option	∼	partial function:
None	∼	no result
Some *a*	∼	result *a*

Example:

```
primrec lookup :: 'k ⇒ ('k × 'v) list ⇒ 'v option
where
  lookup k []       = None |
  lookup k (x #xs) = (if fst x = k then Some (snd x) else lookup k xs)
```

Demo

Section 12

Induction

Structural induction

$P\ xs$ holds for all lists xs if

- P Nil
- and for arbitrary x and xs, $P\ xs \Longrightarrow P\ (x\#xs)$
  Induction theorem **list.induct:**
  $[\![P\ []; \bigwedge a\ list.\ P\ list \Longrightarrow P\ (a\#list)]\!] \Longrightarrow P\ list$
- General proof method for induction: **(induct x)**
  - x must be a free variable in the first subgoal.
  - type of x must be a datatype.

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

Example

A tail recursive list 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

Demo – Proof Attempt

Generalisation

Replace constants by variables

lemma itrev xs ys = rev xs@ys

Quantify free variables by $\forall$
(except the induction variable)

lemma $\forall ys$. itrev xs ys = rev xs@ys

Or: **apply** (induct xs arbitrary: ys)

Exercises

- define a primitive recursive function **lsum** :: nat list $\Rightarrow$ nat that returns the sum of the elements in a list.
- show "$2 * \mathsf{lsum}\ [0..<\mathit{Suc}\ n] = n * (n+1)$"
- show "$\mathsf{lsum}\ (\mathsf{replicate}\ n\ a) = n * a$"
- define a function **lsumT** using a tail recursive version of listsum.
- show that the two functions are equivalent: lsum *xs* = lsumT *xs*

Section 13

General Recursion

General Recursion

The Choice

- Limited expressiveness, automatic termination
  - **primrec**
- High expressiveness, termination proof may fail
  - **fun**
- High expressiveness, tweakable, termination proof manual
  - **function**

fun —Examples

fun sep :: "'a ⇒ 'a list ⇒ 'a list"
where
"sep a (x # y # zs) = x # a # sep a (y # zs)" |
"sep a xs = xs"

fun ack :: "nat ⇒ nat ⇒ nat"
where
"ack 0 n = Suc n" |
"ack (Suc m) 0 = ack m 1" |
"ack (Suc m) (Suc n) = ack m (ack (Suc m) n)"

fun

- More permissive than **primrec**:
  - pattern matching in all parameters
  - nested, linear constructor patterns
  - reads equations sequentially like in Haskell (top to bottom)
  - proves termination automatically in many cases (tries lexicographic order)
- Generates more theorems than **primrec**
- May fail to prove termination:
  - use **function (sequential)** instead
  - allows you to prove termination manually

Demo

fun — Induction Principle

- Each **fun** definition induces an induction principle
- For each equation:

  show P holds for lhs, provided P holds for each recursive call on rhs
- Example **sep.induct**:

$$\begin{array}{l}\llbracket \bigwedge a.\ P\ a\ []; \\ \quad \bigwedge a\ w.\ P\ a\ [w]; \\ \quad \bigwedge a\ x\ y\ zs.\ P\ a\ (y\#zs) \Longrightarrow P\ a\ (x\#y\#zs); \\ \rrbracket \Longrightarrow P\ a\ xs\end{array}$$

Termination

Isabelle tries to prove termination automatically

- For most functions this works with a lexicographic termination relation.
- Sometimes not $\Rightarrow$ error message with unsolved subgoal
- You can prove termination separately.

function (sequential) quicksort **where**
"quicksort [] = []" |
"quicksort (x#xs) = (quicksort [$y \leftarrow xs.\ y \leq x$])@[$x$]@(quicksort [$y \leftarrow xs.\ x < y$])"
by pat_completeness auto

termination
by (relation "measure length") (auto simp: less_Suc_eq_le)

Demo

How does fun/function work?

Recall **primrec**:

- defined one recursion operator per datatype D
- inductive definition of its graph $(x, f\ x) \in D\_rel$
- prove totality: $\forall x.\ \exists y.\ (x, y) \in D\_rel$
- prove uniqueness: $(x, y) \in D\_rel \Rightarrow (x, z) \in D\_rel \Rightarrow y = z$
- recursion operator for datatype D_rec, defined via THE.
- primrec: apply datatype recursion operator

How does fun/function work?

Similar strategy for **fun**:

- a new inductive definition for each **fun** *f*
- extract *recursion scheme* for equations in *f*
- define graph *f_rel* inductively, encoding recursion scheme
- prove totality (= termination)
- prove uniqueness (automatic)
- derive original equations from *f_rel*
- export induction scheme from *f_rel*

How does fun/function work?

function can separate and defer termination proof:

- skip proof of totality
- instead derive equations of the form: $x \in f\_dom \Rightarrow f\ x = \ldots$
- similarly, conditional induction principle
- $f\_dom = acc\ f\_rel$
- acc = accessible part of f_rel
- the part that can be reached in finitely many steps
- termination = $\forall x.\ x \in f\_dom$
- still have conditional equations for partial functions

Demo

Proving Termination

termination fun_name sets up termination goal $\forall x.\ x \in \mathit{fun\_name\_dom}$

Three main proof methods:

- **lexicographic_order** (default tried by **fun**)
- **size_change** (automated translation to simpler size-change graph[1])
- **relation R** (manual proof via well-founded relation)

[1] C.S. Lee, N.D. Jones, A.M. Ben-Amram,
The Size-change Principle for Program Termination, POPL 2001.

Well-Founded Orders

Definition

$<_r$ is well founded if well-founded induction holds

$\mathrm{wf}(<_r) \equiv \forall P.\ (\forall x.\ (\forall y <_r x. P\ y) \longrightarrow P\ x) \longrightarrow (\forall x.\ P\ x)$

Well founded induction rule:

$$\frac{\mathrm{wf}(<_r) \quad \bigwedge x.\ (\forall y <_r x.\ P\ y) \Longrightarrow P\ x}{P\ a}$$

Alternative definition (equivalent):

there are no infinite descending chains, or (equivalent):

every nonempty set has a minimal element wrt $<_r$

$$\begin{array}{lcl} \min\ (<_r)\ Q\ x & \equiv & \forall y \in Q.\ y \not<_r x \\ \mathrm{wf}\ (<_r) & = & (\forall Q \neq \{\}.\ \exists m \in Q.\ \min\ r\ Q\ m) \end{array}$$

Well-Founded Orders: Examples

- $<$ on $\mathbb{N}$ is well founded
  well founded induction = complete induction
- $>$ and $\leq$ on $\mathbb{N}$ are **not** well founded
- $x <_r y = x \text{ dvd } y \wedge x \neq 1$ on $\mathbb{N}$ is well founded
  the minimal elements are the prime numbers
- $(a, b) <_r (x, y) = a <_1 x \vee a = x \wedge b <_2 y$ is well founded
  if $<_1$ and $<_2$ are well founded
- $A <_r B = A \subset B \wedge \text{finite } B$ is well founded
- $\subseteq$ and $\subset$ in general are **not** well founded

More about well founded relations: *Term Rewriting and All That*

Extracting the Recursion Scheme

So far for termination. What about the recursion scheme?

Not fixed anymore as in **primrec**.

Examples:

- **fun** fib **where**
  fib 0 = 1 |
  fib (Suc 0) = 1 |
  fib (Suc (Suc n)) = fib n + fib (Suc n)

  Recursion: Suc (Suc n) $\rightsquigarrow$ n, Suc (Suc n) $\rightsquigarrow$ Suc n
- **fun** f **where** f x = (if x = 0 then 0 else f (x - 1) * 2)

  Recursion: x $\neq$ 0 $\Longrightarrow$ x $\rightsquigarrow$ x - 1

Extracting the Recursion Scheme

Higher Order:

- **datatype** 'a tree = Leaf 'a | Branch 'a tree list

  fun treemap :: ('a ⇒ 'a) ⇒ 'a tree ⇒ 'a tree **where**
  treemap fn (Leaf n) = Leaf (fn n) |
  treemap fn (Branch l) = Branch (map (treemap fn) l)

  Recursion: x ∈ set l ⟹ (fn, Branch l) ⇝ (fn, x)

How does Isabelle extract context information for the call?

Extracting the Recursion Scheme

Extracting context for equations
$\Rightarrow$
Congruence Rules!

Recall rule **if_cong**:

$$\llbracket\, b = c;\ c \Longrightarrow x = u;\ \neg\, c \Longrightarrow y = v \,\rrbracket \Longrightarrow$$
$$(\text{if } b \text{ then } x \text{ else } y) = (\text{if } c \text{ then } u \text{ else } v)$$

Read: for transforming x, use b as context information, for y use $\neg b$.
In fun_def: for recursion in x, use b as context, for y use $\neg b$.

Congruence Rules for fun_defs

The same works for function definitions.

declare my_rule[fundef_cong]
(if_cong already added by default)

Another example (higher-order):
$[\![\text{xs} = \text{ys}; \bigwedge \text{x}.\ \text{x} \in \text{set ys} \Longrightarrow \text{f x} = \text{g x}]\!] \Longrightarrow \text{map f xs} = \text{map g ys}$

Read: for recursive calls in f, f is called with elements of xs

Demo

Further Reading

Alexander Krauss,
Automating Recursive Definitions and Termination Proofs in Higher-Order Logic.
PhD thesis, TU Munich, 2009.

`https://www21.in.tum.de/~krauss/papers/krauss-thesis.pdf`

We have seen ...

- General recursion with **fun**/**function**
- Induction over recursive functions
- How **fun** works
- Termination, partial functions, congruence rules

Section 14

Sledgehammer and Co.

Overview

Automatic Proof and Disproof

- Sledgehammer: automatic proofs
- Quickcheck: counter example by testing
- Nitpick: counter example by SAT

Based on slides by Jasmin Blanchette, Lukas Bulwahn, and Tobias Nipkow (TUM).

Automation

Dramatic improvements in fully automated proofs in the last 2 decades.

- First-order logic (ATP): Otter, Vampire, E, SPASS
- Propositional logic (SAT): MiniSAT, Chaff, RSat
- SAT modulo theory (SMT): CVC3/4/5, Yices, Z3

The key:

Efficient reasoning engines, and restricted logics.

Automation in Isabelle

1980s *rule applications, write ML code*

1990s *simplifier, automatic provers (blast, auto), arithmetic*

2000s *embrace external tools, but don't trust them (ATP/SMT/SAT)*

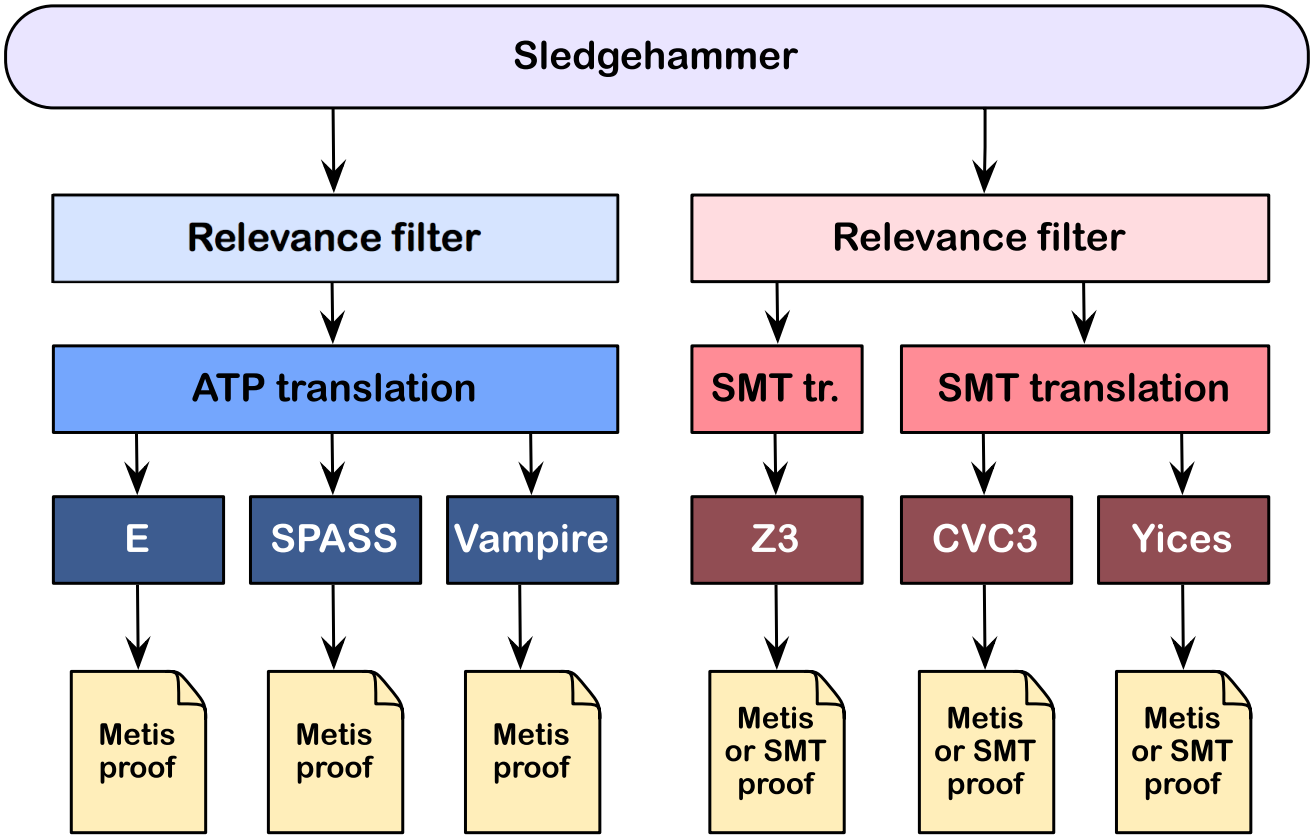
Sledgehammer

Sledgehammer:

- *Connects Isabelle with ATPs and SMT solvers:*
  E, SPASS, Vampire, CVC4, Yices, Z3
- *Simple invocation:*
  - *Users don't need to select or know facts*
  - *or ensure the problem is first-order*
  - *or know anything about the automated prover*
- *Exploits local parallelism and remote servers*

Demo: Sledgehammer

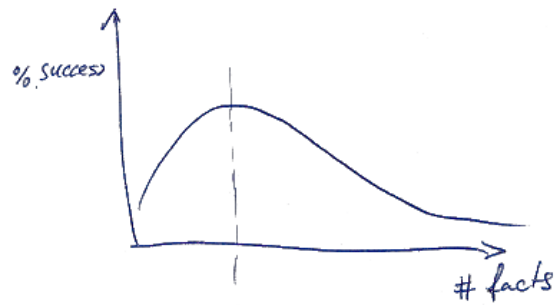
Sledgehammer Architecture



Fact Selection

Provers perform poorly if given 1000s of facts.

- *Best number of facts depends on the prover*
- *Need to take care which facts we give them*
- *Idea: order facts by relevance, give top n to prover*
  ($n = 250, 1000, \ldots$)
- *Meng & Paulson method: lightweight, symbol-based filter*
- *Machine learning method:*
  look at previous proofs to get a probability of relevance



From HOL to FOL

Source: *higher-order, polymorphism, type classes*
Target: *first-order, untyped or simply-typed*

- ***First-order:***
  - *SK combinators, λ-lifting*
  - *Explicit function application operator*
- ***Encode types:***
  - *Monomorphise (generate multiple instances), or*
  - *Encode polymorphism on term level*

Reconstruction

We don't want to trust the external provers.
Need to check/reconstruct proof.

- *Re-find using Metis*
  Usually fast and reliable (sometimes too slow)
- *Rerun external prover for trusted replay*
  Used for SMT. Re-runs prover each time!
- *Recheck stored explicit external representation of proof*
  Used for SMT, no need to re-run. Fragile.
- *Recast into structured Isar proof*
  Fast, not always readable.

Judgement Day (up to 2013)

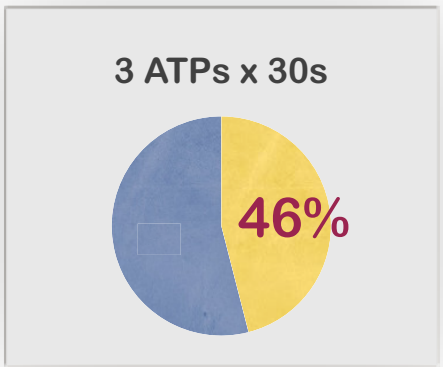
Evaluating Sledgehammer:

- *1240 goals out of 7 existing theories.*
- *How many can sledgehammer solve?*

- ***2010:*** *E, SPASS, Vampire (for 5-120s). 46%*
  $ESV \times 5s \approx V \times 120s$
- ***2011:*** *Add E-SInE, CVC2, Yices, Z3 (30s).*
  $Z3 > V$
- ***2012:*** *Better integration with SPASS. 64%*
  SPASS best (small margin)
- ***2013:*** *Machine learning for fact selection. 69%*
  Improves a few percent across provers.

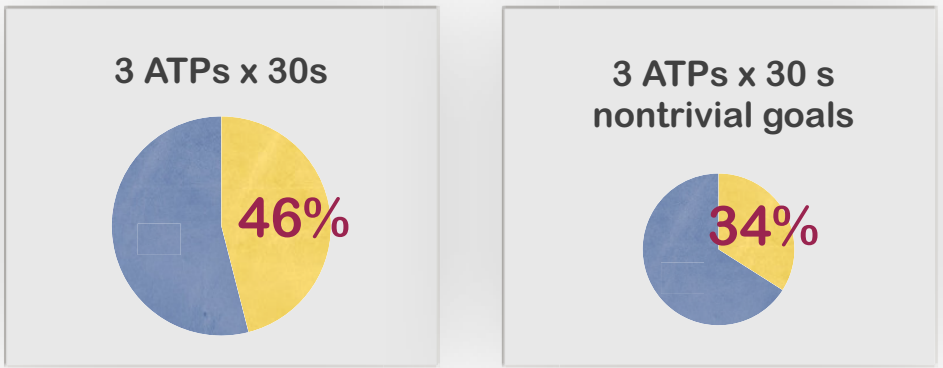
Evaluation

2010

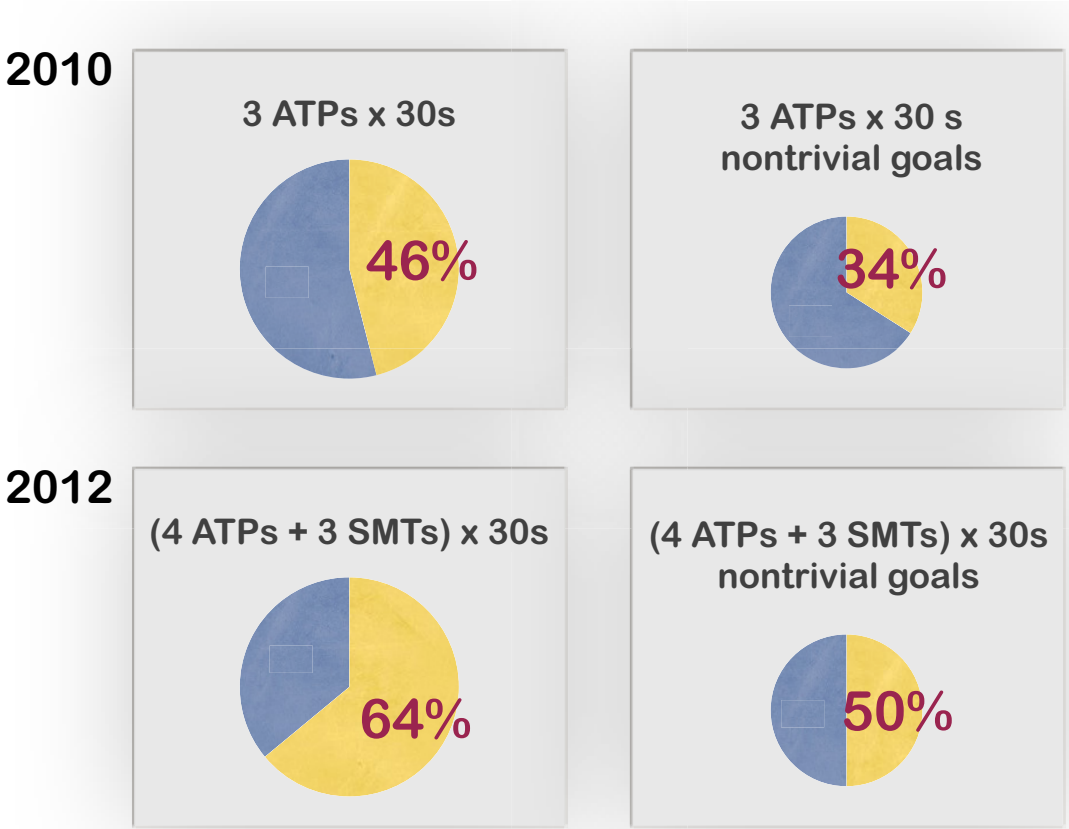


Evaluation

2010



Evaluation


2010
3 ATPs x 30s
46%
3 ATPs x 30 s nontrivial goals
34%
2012
(4 ATPs + 3 SMTs) x 30s
64%
(4 ATPs + 3 SMTs) x 30s nontrivial goals
50%

Judgement Day (2016)

Prover	MePo	MaSh	MeSh	Any selector
CVC4 1.5pre	679	749	**783**	830
E 1.8	622	601	**665**	726
SPASS 3.8ds	678	684	**739**	789
Vampire 3.0	703	698	**740**	789
veriT 2014post	543	556	**590**	655
Z3 4.3.2pre	638	668	**703**	788
Any prover	801	885	**919**	943

Fig. 15 Number of successful Sledgehammer invocations per prover on 1230 Judgment Day goals

919/1230 = **74%**

Sledgehammer rules!

Example application:

- *Large Isabelle/HOL repository of algebras for modelling imperative programs*
  (Kleene Algebra, Hoare logic, ..., ≈ 1000 lemmas)
- *Intricate refinement and termination theorems*
- *Sledgehammer and Z3 automate algebraic proofs at textbook level.*

"The integration of ATP, SMT, and Nitpick is for our purposes very very helpful."

Disproof

Theorem proving and testing

Testing can show only the presence of errors, but not their absence. (Dijkstra)

Testing cannot prove theorems, but it can refute conjectures!

Sad facts of life:

- *Most lemma statements are wrong the first time.*
- *Theorem proving is expensive as a debugging technique.*

Find counter examples automatically!

Quickcheck

Lightweight validation by testing.

- *Motivated by Haskell's QuickCheck*
- *Uses Isabelle's code generator*
- *Fast*
- *Runs in background, proves you wrong as you type.*

Quickcheck

Covers a number of testing approaches:

- Random and exhausting testing.
- Smart test data generators.
- Narrowing-based (symbolic) testing.

Creates test data generators automatically.

Demo: Quickcheck

Test generators for datatypes

Fast iteration in continuation-passing-style

datatype α list = Nil | Cons α (α list)

Test function:

$$\text{test}_{\alpha\ \textit{list}}\ \text{P} = \text{P Nil}\ \textit{andalso}\ \text{test}_{\alpha}\ (\lambda\text{x. test}_{\alpha\ \textit{list}}\ (\lambda\text{xs. P (Cons x xs)}))$$

Test generators for predicates

$$\text{distinct xs} \Longrightarrow \text{distinct (remove1 x xs)}$$

Problem:
Exhaustive testing creates many useless test cases.

Solution:
Use definitions in precondition for smarter generator.
Only generate cases where distinct xs is true.

test-distinct$_{\alpha\ list}$ *P = P Nil andalso*
test$_{\alpha}$ *(*λ*x. test-distinct*$_{\alpha\ list}$ *(if x* $\notin$ *xs then (*λ*xs. P (Cons x xs)) else True))*

Use data flow analysis to figure out which variables must be computed and which generated.

Narrowing

Symbolic execution with demand-driven refinement

- *Test cases can contain variables*
- *If execution cannot proceed: instantiate with further symbolic terms*

Pays off if large search spaces can be discarded:

distinct (Cons 1 (Cons 1 x))

False for any x, no further instantiations for x necessary.

Implementation:
Lazy execution with outer refinement loop.
Many re-computations, but fast.

Quickcheck Limitations

Only executable specifications!

- *No equality on functions with infinite domain*
- *No axiomatic specifications*

Nitpick

Nitpick

Finite model finder

- *Based on SAT via Kodkod (backend of Alloy prover)*
- *Soundly approximates infinite types*

Nitpick Successes

- *Algebraic methods*
- *C++ memory model*
- *Found soundness bugs in TPS and LEO-II*

Fan mail:

"Last night I got stuck on a goal I was sure was a theorem. After 5–10 minutes I gave Nitpick a try, and within a few secs it had found a splendid counterexample—despite the mess of locales and type classes in the context!"

Demo: Nitpick

Automation Summary

- Proof: Sledgehammer
- Counter examples: Quickcheck
- Counter examples: Nitpick

Section 15

Isar (Part 2)

Datatypes in Isar

General Case Distinctions

show *formula*
proof -
  have $P_1 \vee P_2 \vee P_3$ <proof>
  moreover { **assume** P_1 ... **have** ?thesis <proof> }
  moreover { **assume** P_2 ... **have** ?thesis <proof> }
  moreover { **assume** P_3 ... **have** ?thesis <proof> }
  ultimately show ?thesis **by** blast
qed

{ . . . } is a proof block similar to **proof** ... **qed**

{ **assume** P_1 . . . **have** P <proof> }
stands for $P_1 \Longrightarrow P$

Datatype case distinction

proof (cases *term*)
  case Constructor_1
  ⋮
next
⋮
next
  case (Constructor_k $\vec{x}$)
  $\ldots$ $\vec{x}$ $\ldots$
qed

case (Constructor_i $\vec{x}$) $\equiv$
fix $\vec{x}$ **assume** Constructor_i : "$term = \text{Constructor}_i$ $\vec{x}$"

Structural induction for nat

show $P\ n$
proof (induct n)
  case 0 $\equiv$ **let** ?*case* = $P\ 0$
  ...
  show ?*case*
next
  case (Suc n) $\equiv$ **fix** n **assume** Suc: $P\ n$
  ... **let** ?*case* = P (Suc n)
  ... n ...
  show ?*case*
qed

Structural induction: $\Longrightarrow$ and $\bigwedge$

show "$\bigwedge x.\ A\ n \Longrightarrow P\ n$"
proof (induct n)
  case 0
  ...
  show ?*case*
next
  case (Suc n)
  ...
  ... n ...
  ...
  show ?*case*
qed

$\equiv$ **fix** x **assume** 0: "$A\ 0$"
let ?*case* = "$P\ 0$"

$\equiv$ **fix** n and x
assume Suc: "$\bigwedge x.\ A\ n \Longrightarrow P\ n$"
"A (Suc n)"
let ?*case* = "P (Suc n)"

Demo: Datatypes in Isar

Calculational Reasoning

The Goal

Prove:

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

using:

assoc: $(x \cdot y) \cdot z = x \cdot (y \cdot z)$

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

left_one: $1 \cdot x = x$

The Goal

Prove:

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

assoc:	$(x \cdot y) \cdot z = x \cdot (y \cdot z)$
left_inv:	$x^{-1} \cdot x = 1$
left_one:	$1 \cdot x = x$

Can we do this in Isabelle?

- Simplifier: too eager
- Manual: difficult in apply style
- Isar: with the methods we know, too verbose

Chains of equations

The Problem

$$\begin{array}{lcl} a & = & b \\ \ldots & = & c \\ \ldots & = & d \end{array}$$

shows $a = d$ by transitivity of $=$

Each step usually nontrivial (requires own subproof)

Solution in Isar:

- Keywords **also** and **finally** to delimit steps
- **...**: predefined schematic term variable, refers to right hand side of last expression
- Automatic use of transitivity rules to connect steps

also/finally

have “$t_0 = t_1$” [proof]
also
have “... $= t_2$” [proof]
also
$\vdots$
also
have “$\cdots = t_n$” [proof]
finally
show P
— ’finally’ pipes fact “$t_0 = t_n$” into the proof

calculation register
“$t_0 = t_1$”

“$t_0 = t_2$”

$\vdots$

“$t_0 = t_{n-1}$”

$t_0 = t_n$

More about also

- Works for all combinations of $=$, $\leq$ and $<$.
- Uses all rules declared as `[trans]`.
- To view all combinations: `print_trans_rules`

Designing [trans] Rules

have = "$l_1 \odot r_1$" [proof]
also
have "$\ldots \odot r_2$" [proof]
also

Anatomy of a [trans] rule:

- Usual form: plain transitivity $[\![l_1 \odot r_1; r_1 \odot r_2]\!] \Longrightarrow l_1 \odot r_2$
- More general form: $[\![P\ l_1\ r_1; Q\ r_1\ r_2; A]\!] \Longrightarrow C\ l_1\ r_2$

Examples:

- pure transitivity: $[\![a = b; b = c]\!] \Longrightarrow a = c$
- mixed: $[\![a \leq b; b < c]\!] \Longrightarrow a < c$
- substitution: $[\![P\ a; a = b]\!] \Longrightarrow P\ b$
- antisymmetry: $[\![a < b; b < a]\!] \Longrightarrow \mathit{False}$
- monotonicity:
  $[\![a = f\ b; b < c; \bigwedge x\ y.\ x < y \Longrightarrow f\ x < f\ y]\!] \Longrightarrow a < f\ c$

Demo

Finding Theorems

Command **find_theorems** (C-c C-f) finds combinations of:

- pattern: ”_ **+** _ **+** _”
- lhs of simp rules: **simp:** ”_ ***** **(**_ **+** _**)**”
- intro/elim/dest on current goal
- lemma name: **name: assoc**
- exclusions thereof: **-name:** ”**HOL.**”

find_theorems dest -”hd” name: ”List.”

finds all theorems in the current context that

- match the goal as dest rule,
- do not contain the constant ”hd”
- are in the List theory (name starts with ”List.”)

Isar: define and defines

Can define vnameal constant in Isar proof context:

proof

...

define ”f ≡ big term”

have ”g = f x” ...

like definition, not automatically unfolded (f_def)

different to **let** ?f = ”big term”

Also available in lemma statement:

lemma ...:

fixes ...

assumes ...

defines ...

shows ...

Section 16

Floyd-Hoare Logic

Semantics (A Crash Course)

Further Details

- see Concrete Semantics
- COMP3610/6361 Principles of Programming Languages
  `https://comp.anu.edu.au/courses/comp3610/`

IMP - a small Imperative Language

Commands:

datatype com	=	SKIP	
	|	Assign vname aexp	(_ := _)
	|	Semi com com	(_ ; _)
	|	Cond bexp com com	(IF _ THEN _ ELSE _)
	|	While bexp com	(WHILE _ DO _ OD)
type_synonym vname	=	string	
type_synonym state	=	vname ⇒ nat	
type_synonym aexp	=	state ⇒ nat	
type_synonym bexp	=	state ⇒ bool	

Example Program

Usual syntax:

$$
\begin{array}{l}
B := 1; \\
\textsf{WHILE } A \neq 0 \textsf{ DO} \\
\quad B := B * A; \\
\quad A := A - 1 \\
\textsf{OD}
\end{array}
$$

Expressions are functions from state to bool or nat:

$$
\begin{array}{l}
B := (\lambda\sigma.\ 1); \\
\textsf{WHILE } (\lambda\sigma.\ \sigma\ A \neq 0) \textsf{ DO} \\
\quad B := (\lambda\sigma.\ \sigma\ B * \sigma\ A); \\
\quad A := (\lambda\sigma.\ \sigma\ A - 1) \\
\textsf{OD}
\end{array}
$$

What does it do?

So far we have defined:

- Syntax of commands and expressions
- State of programs (function from variables to values)

Now we need: the meaning (semantics) of programs

How to define execution of a program?

- A wide field of its own
- Some choices:
  - Operational (inductive relations, big step, small step)
  - Denotational (programs as functions on states, state transformers)
  - Axiomatic (pre-/post conditions, Hoare logic)

Structural Operational Semantics

$$\frac{}{\langle \mathsf{SKIP}, \sigma \rangle \to \sigma}$$

$$\frac{e\ \sigma = v}{\langle \mathsf{x} := \mathsf{e}, \sigma \rangle \to \sigma[x \mapsto v]}$$

$$\frac{\langle c_1, \sigma \rangle \to \sigma' \quad \langle c_2, \sigma' \rangle \to \sigma''}{\langle c_1; c_2, \sigma \rangle \to \sigma''}$$

$$\frac{b\ \sigma = \mathsf{True} \quad \langle c_1, \sigma \rangle \to \sigma'}{\langle \mathsf{IF}\ b\ \mathsf{THEN}\ c_1\ \mathsf{ELSE}\ c_2, \sigma \rangle \to \sigma'}$$

$$\frac{b\ \sigma = \mathsf{False} \quad \langle c_2, \sigma \rangle \to \sigma'}{\langle \mathsf{IF}\ b\ \mathsf{THEN}\ c_1\ \mathsf{ELSE}\ c_2, \sigma \rangle \to \sigma'}$$

Structural Operational Semantics

$$\frac{b\ \sigma = \text{False}}{\langle \text{WHILE } b \text{ DO } c \text{ OD}, \sigma \rangle \to \sigma}$$

$$\frac{b\ \sigma = \text{True} \quad \langle c, \sigma \rangle \to \sigma' \quad \langle \text{WHILE } b \text{ DO } c \text{ OD}, \sigma' \rangle \to \sigma''}{\langle \text{WHILE } b \text{ DO } c \text{ OD}, \sigma \rangle \to \sigma''}$$

Demo: The Definitions in Isabelle

Proofs about Programs

Now we know:

- What programs are: Syntax
- On what they work: State
- How they work: Semantics

So we can prove properties about programs

Example:
Show that example program from earlier implements the factorial.

$$\textbf{lemma}\ \langle \text{factorial}, \sigma \rangle \to \sigma' \Longrightarrow \sigma' B = \text{fac}\ (\sigma A)$$
$$(\text{where} \quad \text{fac}\ 0 = 1, \quad \text{fac}\ (\text{Suc}\ n) = (\text{Suc}\ n) * \text{fac}\ n)$$

Demo: Example Proof

Too tedious

Induction needed for each loop

Is there something easier?

Floyd-Hoare Logic

Floyd-Hoare Logic

Idea: describe meaning of program by pre/post conditions

Examples:

$\{\mathsf{True}\} \quad x := 2 \quad \{x = 2\}$

$\{y = 2\} \quad x := 21 * y \quad \{x = 42\}$

$\{x = n\} \quad \text{IF } y < 0 \text{ THEN } x := x + y \text{ ELSE } x := x - y \quad \{x = n - |y|\}$

$\{A = n\} \quad \mathsf{factorial} \quad \{B = \mathsf{fac}\ n\}$

Proofs: have rules that directly work on such triples

Meaning of a Hoare-Triple

$$\{P\} \quad c \quad \{Q\}$$

What are the assertions P and Q?

- Here: again functions from state to bool (shallow embedding of assertions)
- Other choice: syntax and semantics for assertions (deep embedding)

What does $\{P\}\ c\ \{Q\}$ mean?

Partial Correctness:

$$\models \{P\}\ c\ \{Q\} \quad \equiv \quad \forall \sigma\ \sigma'.\ P\ \sigma \wedge \langle c, \sigma \rangle \to \sigma' \longrightarrow Q\ \sigma'$$

Total Correctness:

$$\begin{aligned} \models \{P\}\ c\ \{Q\} \quad \equiv \quad & (\forall \sigma\ \sigma'.\ P\ \sigma \wedge \langle c, \sigma \rangle \to \sigma' \longrightarrow Q\ \sigma') \wedge \\ & (\forall \sigma.\ P\ \sigma \longrightarrow \exists \sigma'.\ \langle c, \sigma \rangle \to \sigma') \end{aligned}$$

This lecture: partial correctness only (easier)

Hoare Rules

$$\frac{}{\{P\}\quad \text{SKIP}\quad \{P\}} \qquad \frac{}{\{P[x \mapsto e]\}\quad x := e\quad \{P\}}$$

$$\frac{\{P\}\ c_1\ \{R\} \quad \{R\}\ c_2\ \{Q\}}{\{P\}\quad c_1; c_2\quad \{Q\}}$$

$$\frac{\{P \wedge b\}\ c_1\ \{Q\} \quad \{P \wedge \neg b\}\ c_2\ \{Q\}}{\{P\}\quad \text{IF } b \text{ THEN } c_1 \text{ ELSE } c_2\quad \{Q\}}$$

$$\frac{\{P \wedge b\}\ c\ \{P\} \quad P \wedge \neg b \Longrightarrow Q}{\{P\}\quad \text{WHILE } b \text{ DO } c \text{ OD}\quad \{Q\}}$$

$$\frac{P \Longrightarrow P' \quad \{P'\}\ c\ \{Q'\} \quad Q' \Longrightarrow Q}{\{P\}\quad c\quad \{Q\}}$$

Hoare Rules

$$\frac{}{\vdash \{P\} \quad \text{SKIP} \quad \{P\}} \qquad \frac{}{\vdash \{\lambda\sigma.\ P\ (\sigma(x := e\ \sigma))\} \quad x := e \quad \{P\}}$$

$$\frac{\vdash \{P\}\ c_1\ \{R\} \quad \vdash \{R\}\ c_2\ \{Q\}}{\vdash \{P\} \quad c_1; c_2 \quad \{Q\}}$$

$$\frac{\vdash \{\lambda\sigma.\ P\ \sigma \wedge b\ \sigma\}\ c_1\ \{Q\} \quad \vdash \{\lambda\sigma.\ P\ \sigma \wedge \neg b\ \sigma\}\ c_2\ \{Q\}}{\vdash \{P\} \quad \text{IF } b \text{ THEN } c_1 \text{ ELSE } c_2 \quad \{Q\}}$$

$$\frac{\vdash \{\lambda\sigma.\ P\ \sigma \wedge b\ \sigma\}\ c\ \{P\} \quad \bigwedge\sigma.\ P\ \sigma \wedge \neg b\ \sigma \Longrightarrow Q\ \sigma}{\vdash \{P\} \quad \text{WHILE } b \text{ DO } c \text{ OD} \quad \{Q\}}$$

$$\frac{\bigwedge\sigma.\ P\ \sigma \Longrightarrow P'\ \sigma \quad \vdash \{P'\}\ c\ \{Q'\} \quad \bigwedge\sigma.\ Q'\ \sigma \Longrightarrow Q\ \sigma}{\vdash \{P\} \quad c \quad \{Q\}}$$

Are the Rules Correct?

Soundness: $\vdash \{P\}\ c\ \{Q\} \Longrightarrow \models \{P\}\ c\ \{Q\}$

Proof: by rule induction on $\vdash \{P\}\ c\ \{Q\}$

Demo: Hoare Logic in Isabelle

We have seen ...

- Syntax of a simple imperative language
- Operational semantics
- Program proof on operational semantics
- Hoare logic rules
- Soundness of Hoare logic

Automation?

Hoare rule application is nicer than using operational semantics.

BUT:

- it's still kind of tedious
- it seems boring & mechanical

Automation?

Invariant

Problem: While – need creativity to find right (invariant) P

Solution:

- annotate program with invariants
- then, Hoare rules can be applied automatically

Example:

$\{M = 0 \wedge N = 0\}$
WHILE $M \neq a$ INV $\{N = M * b\}$ DO $N := N + b; M := M + 1$ OD
$\{N = a * b\}$

Weakest Preconditions

pre $c\ Q$ **= weakest** P **such that** $\{P\}\ c\ \{Q\}$

With annotated invariants, easy to get:

$$
\begin{array}{lcl}
\text{pre SKIP } Q & = & Q \\
\text{pre } (x := a)\ Q & = & \lambda\sigma.\ Q(\sigma(x := a\sigma)) \\
\text{pre } (c_1; c_2)\ Q & = & \text{pre } c_1\ (\text{pre } c_2\ Q) \\
\text{pre (IF } b \text{ THEN } c_1 \text{ ELSE } c_2)\ Q & = & \lambda\sigma.\ (b\sigma \longrightarrow \text{pre } c_1\ Q\ \sigma) \wedge \\
 & & \quad (\neg b\sigma \longrightarrow \text{pre } c_2\ Q\ \sigma) \\
\text{pre (WHILE } b \text{ INV } I \text{ DO } c \text{ OD})\ Q & = & I
\end{array}
$$

Verification Conditions

$\{\text{pre } c\ Q\}\ c\ \{Q\}$ **only true under certain conditions**

These are called verification conditions vc $c\ Q$:

$$
\begin{array}{lcl}
\text{vc SKIP } Q & = & \text{True} \\
\text{vc } (x := a)\ Q & = & \text{True} \\
\text{vc } (c_1; c_2)\ Q & = & \text{vc } c_2\ Q \wedge (\text{vc } c_1\ (\text{pre } c_2\ Q)) \\
\text{vc (IF } b \text{ THEN } c_1 \text{ ELSE } c_2)\ Q & = & \text{vc } c_1\ Q \wedge \text{vc } c_2\ Q \\
\text{vc (WHILE } b \text{ INV } I \text{ DO } c \text{ OD})\ Q & = & (\forall \sigma.\ I\sigma \wedge b\sigma \longrightarrow \text{pre } c\ I\ \sigma) \wedge \\
 & & (\forall \sigma.\ I\sigma \wedge \neg b\sigma \longrightarrow Q\ \sigma) \wedge \\
 & & \text{vc } c\ I
\end{array}
$$

$$\text{vc } c\ Q \wedge (P \Longrightarrow \text{pre } c\ Q) \Longrightarrow \{P\}\ c\ \{Q\}$$

Syntax Tricks

- $x := \lambda\sigma.\ 1$ instead of $x := 1$ sucks
- $\{\lambda\sigma.\ \sigma\ x = n\}$ instead of $\{x = n\}$ sucks as well

Problem: program variables are functions, not values

Solution: distinguish program variables syntactically

Choices:

- declare program variables with each Hoare triple
  - nice, usual syntax
  - works well if you state full program and only use **vcg**
- separate program variables from Hoare triple (ext. records), indicate usage as function syntactically
  - more syntactic overhead
  - program pieces compose nicely

Demo

Arrays

Depending on language, model arrays as functions:

- Array access = function application:
  a[i] $=$ a i
- Array update = function update:
  a[i] :== v $=$ a :== a(i:= v)

Use lists to express length:

- Array access = nth:
  a[i] $=$ a ! i
- Array update = list update:
  a[i] :== v $=$ a :== a[i:= v]
- Array length = list length:
  a.length $=$ length a

Pointers

Choice 1

datatype ref = Ref int | Null
types heap = int ⇒ val
datatype val = Int int | Bool bool | Struct_x int int bool | ...

- hp :: heap, p :: ref
- Pointer access: *p = the_Int (hp (the_addr p))
- Pointer update: *p :== v = hp :== hp ((the_addr p) := v)

- a bit clunky
- gets even worse with structs
- lots of value extraction (the_Int) in spec and program

Pointers

Choice 2 (Burstall '72, Bornat '00)

Example: struct with next pointer and element

datatype	ref	= Ref int | Null
types	next_hp	= int $\Rightarrow$ ref
types	elem_hp	= int $\Rightarrow$ int

- next :: next_hp, elem :: elem_hp, p :: ref
- Pointer access: p$\rightarrow$next $=$ next (the_addr p)
- Pointer update: p$\rightarrow$next :== v $=$ next :== next ((the_addr p) := v)

In general:

- a separate heap for each struct field
- buys you p$\rightarrow$next $\neq$ p$\rightarrow$elem automatically (aliasing)
- still assumes type safe language

Demo

We have seen ...

- Weakest precondition
- Verification conditions
- Example program proofs
- Arrays, pointers