Preuves Interactives et Applications

Burkhart Wolff

http://www.lri.fr/~wolff/teach-material/2018-19/M2-CSMR

Université Paris-Saclay

Introduction to λ-calculus

Motivation: Why ITP ?

- Program verification:
  - SEL4 (Isabelle/HOL, NICTA), secured micro-kernel for OS
  - Compcert (Coq, Inria), optimizing C compiler
  - Security : moderlling of JavaCard plateforms
  - Mathematics : 4 color theorem, Kepler conjecture,
    Feit-Thompson conjecture. . .
  - Formal proofs in informatics
  - machine arithmetics (nombres flottants)
  - crypt algorithms, combinatory algorithms
  - program language semantics
  - Back-end for other provers (reverifying proof traces),
  - proof obligations in program verification
  - test-case generations
  - … much stuff in Phd-thesis and the scientific literature …

Plan of this Course

- The „λ-calculus"
- α-conversion,β-reduction,ε-reduction
- What is „typed λ-calculus"
- Using typed λ-calculus to represent logical systems
- What is „natural deduction" ?
(from another perspective)

Foundation: The λ-calculus

- Developed in the 30ies by Alonzo Church (and his students Kleene and Rosser)
- … to develop a representation of Whitehead's and Russel's „Principia Mathematica"
- … was early on detected as Turing-complete and actually a "functional computation model" (Turing)

The λ-calculus

- The „Pure λ-calculus" : a term language. λ-terms T are built (inductively) over:
  - V, a set of "variable symbols"
  - λV. T, a term construction called "λ-abstraction" ,
  - T T , a term construction called "λ-abstraction"

- A version adding a set of constant symbols is called „the applied λ-calculus"

The λ-calculus

This produces expressions like:

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

parenthesis can be dropped:

$((f\ x)\ y)$ is written just $f\ x\ y$

$f(x)$ is written just $f\ x$.

The λ-calculus

The most important aspect of „variables" are that they „stand for something", i.e. they can be „substituted" by something.

A key-motivation for the λ-calculus is that key-ideas of binding and scoping of variables (as occurring mathematics and programming languages) should be treated correctly.

λ-abstractions build a scope: in λx. x x, x appears "bound". If a variable occurrence in not bound, is is called "free".

Plan of this Course

- The „λ-calculus"
- α-conversion,β-reduction,ε-reduction
- What is „typed λ-calculus"
- Using typed λ-calculus to represent logical systems
- What is „natural deduction" ?
  (from another perspective)

The λ-calculus

Example:

(λx.λy.(λz.(λx.z a) (λy.z y)) (x y))

The free variables can be computed recursively:

free(x) = {x} for any x ∈ V

free(T T') = free(T) ∪ free(T')

free(λx. T) = free(T) \ {x}

Substitution and Conversions

Bound variables can be arbitrarily renamed, provided that this does not "capture" a free variable (make it bound). This is reflected by the notion of

α-conversion (written $\leftrightarrow_\alpha$).

Example:

$(\lambda x.\lambda y.(\lambda z.(\lambda x.z\ a)\ (\lambda y.z\ y))\ (x\ y)) \leftrightarrow_\alpha$

$(\lambda x.\lambda y.(\lambda z.(\lambda y.z\ a)\ (\lambda y.z\ y))\ (x\ y))$ but not:

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

Substitution and Conversions

Free-ness of variables and $\leftrightarrow_\alpha$ together give a notion of capture-free substitution.

- $x[x:=r] = r$
- $y[x:=r] = y$
- $(ts)[x:=r] = (t[x:=r])(s[x:=r])$
- $(\lambda x.t)[x:=r] = \lambda x.t$
- $(\lambda y.t)[x:=r] = \lambda y.(t[x:=r])$ if $x \neq y$ and y is not in the free variables of r. The variable y is said to be "fresh" for r.

Substitution and Conversions

Example:

- $(\lambda x.x)[y:=y] = \lambda x.(x[y:=y]) = \lambda x.x$
- $((\lambda x.y)x)[x:=y] = ((\lambda x.y)[x:=y])(x[x:=y]) = (\lambda x.y)\ y$

- Counterexample (ignoring freshness condition) :

$$(\lambda x.y)[y:=x]=\lambda x.(y[y:=x])=\lambda x.x$$

so we would convert a constant function into an identity ...

Substitution and Conversions

The "Motor" of the λ-calculus: the β-conversion (written $\leftrightarrow_\beta$) or its one-directional version, the β-reduction (written $\rightarrow_\beta$). It captures the notion of applying functions to their arguments:

- $(\lambda x.t)\ E \leftrightarrow_\beta t[x:=E]$
- $(\lambda x.t)\ E \rightarrow_\beta t[x:=E]$

Substitution and Conversions

The η-conversion (written $\leftrightarrow_\eta$) or its one-directional version, the η-reduction (written $\rightarrow_\eta$) captures the notion of extensionality on functions:

- $(\lambda x.f\ x) \leftrightarrow_\eta f$ where x does not occur free in f
- $(\lambda x.f\ x) \rightarrow_\eta f$ where x does not occur free in f

All conversions/reductions are congruences, i.e. can be applied to any subterm.

Substitution and Conversions

Example:

$\lambda g.(\lambda x.g\ (x\ x))\ (\lambda x.g\ (x\ x))$ (which we will abbreviate Y)

Now consider:

$$
\begin{array}{ll}
 & \mathbf{Y}\ f \\
\equiv & (\lambda h.(\lambda x.h\ (x\ x))\ (\lambda x.h\ (x\ x)))\ f \\
\rightarrow_\beta & (\lambda x.f\ (x\ x))\ (\lambda x.f\ (x\ x)) \\
\rightarrow_\beta & f\ ((\lambda x.f\ (x\ x))\ (\lambda x.f\ (x\ x))) \\
\equiv & f\ (\mathbf{Y}\ f)
\end{array}
$$

A combinator with this property $\mathbf{Y}\ f = f\ (\mathbf{Y}\ f)$ is called fixpoint combinator.

Substitution and Conversions

Example:

$\lambda g.(\lambda x.g\ (x\ x))\ (\lambda x.g\ (x\ x))$ (which we will abbreviate Y)

Now consider:

$$
\begin{array}{ll}
 & \mathbf{Y}\ f \\
\equiv & (\lambda h.(\lambda x.h\ (x\ x))\ (\lambda x.h\ (x\ x)))\ f \\
\to_\beta & (\lambda x.f\ (x\ x))\ (\lambda x.f\ (x\ x)) \\
\to_\beta & f\ ((\lambda x.f\ (x\ x))\ (\lambda x.f\ (x\ x))) \\
\equiv & f\ (\mathbf{Y}\ f)
\end{array}
$$

A combinator with this property $\mathbf{Y}\ f = f\ (\mathbf{Y}\ f)$ is called fixpoint combinator.

Substitution and Conversions

Example:

$0 \equiv \lambda f.\lambda x.\ x$
$1 \equiv \lambda f.\lambda x. f\ x$
$2 \equiv \lambda f.\lambda x. f\ (f\ x)$
$3 \equiv \lambda f.\lambda x. f\ (f\ (f\ x))$
...

$\mathrm{SUCC} \equiv \lambda n.\lambda f.\lambda x. f\ (n\ f\ x)$

$\mathrm{PLUS} \equiv \lambda m.\lambda n.\lambda f.\lambda x. m\ f\ (n\ f\ x)$

Consider:

$\mathrm{PLUS}\ 2\ 3 \quad \rightarrow_{\beta}^{*} \quad 5$

Substitution and Conversions

Example (Church Numerals):

$0 \equiv \lambda f.\lambda x.\ x$
$1 \equiv \lambda f.\lambda x.f\ x$
$2 \equiv \lambda f.\lambda x.f\ (f\ x)$
$3 \equiv \lambda f.\lambda x.f\ (f\ (f\ x))$
...

$SUCC \equiv \lambda n.\lambda f.\lambda x.f\ (n\ f\ x)$
$PLUS \equiv \lambda m.\lambda n.\lambda f.\lambda x.m\ f\ (n\ f\ x)$
$MULT \equiv \lambda m.\lambda n.\lambda f.m\ (n\ f)$

Consider:

$$\text{PLUS } 2\ 3 \quad \rightarrow_\beta^* \quad 5$$

Substitution and Conversions

Example (Boolean Logics):

TRUE ≡ $\lambda x.\lambda y.x$

FALSE ≡ $\lambda x.\lambda y.y$ (Note that FALSE is equivalent to the Church numeral zero defined above)

AND ≡ $\lambda p.\lambda q.p\ q\ p$

OR ≡ $\lambda p.\lambda q.p\ p\ q$

NOT ≡ $\lambda p.$p FALSE TRUE

IFTHENELSE ≡ $\lambda p.\lambda a.\lambda b.p\ a\ b$

Consider:

AND TRUE FALSE $\rightarrow_\beta^*$ FALSE

Substitution and Conversions

Example (Recursive Function):

FAC ≡ λ*fac*. λ*n*. *IFTHENELSE (ISZERO n)(1) (MULT n (fac(PRED n)))*
Y ≡ λf. (λx. f(x x)) (λx. f(x x))

Consider:

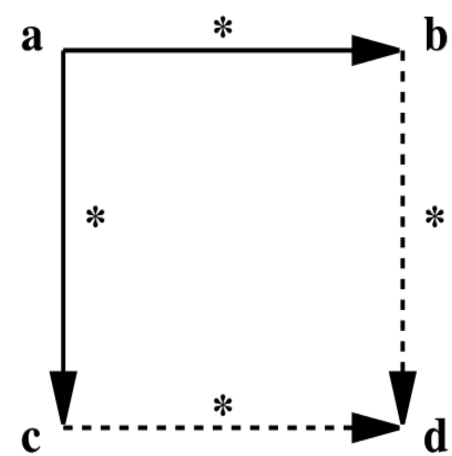
$$(Y\ FAC)\ 4 \quad \rightarrow_{\beta}^{*} \quad 24$$

The untyped λ-calculus

Theoretical Properties (Pure/Applied)

- it is “a universal language“ (i.e. it has the same computational power than, say, Turing Machines
- there may be calculations that „diverge“ (loop)
- it is Church-Rosser:

(for * be β reductions, αη-conversions)



- the equality on λ-*terms* is undecidable.
- the difference between “Pure” and “Applied” irrelevant

Plan of this Course

- The „λ-calculus"
- α-conversion,β-reduction,ε-reduction
- What is „typed λ-calculus"
- Using typed λ-calculus to represent logical systems
- What is „natural deduction" ? (from another perspective)

The typed λ-calculus

Motivation:

- a term - language for representing maths (with integrals, limits and stuff – thus: variables and binding.) in a logic [seminal paper by Church in 1940]
- no divergence admissible [what would a „divergent term" mean in a logic ?]
- equality on terms decidable
- turned out to be easy to implement.

The typed λ-calculus

Idea:

- we use an applied λ-calculus (and constant symbols will be subtly different from variables in the typed λ)
- we introduce the syntactic category of types
- we require all „legal“ terms to be typed, i.e. an association of a term to a type according to typing rules must be possible.

The typed λ-calculus

Types (1):

- We assume a set of type constructors χ with symbols like bool, nat, int, _list, _set, _⇒_, ...
- For type constructors (and constant symbols), we will allow infix/circumfix notation:

  we will write:

nat list	for	(list_)(nat)
bool ⇒ nat	for	(_⇒_)(bool, nat)
. . .		

The typed λ-calculus

Types (1):

- The set of types τ is inductively defined:

$$\tau ::= TV \mid \chi(\tau_1,\ldots,\tau_n)$$

where TV is a set of type variables α,β,γ.
Note: For nat() we just write nat.

The typed λ-calculus

Types (2):

- A C-environment which assigns each constant symbol a type:

  $\Sigma :: C \mapsto \tau$

- A V-environment which assigns to each variable symbol a type:

  $\Gamma :: V \mapsto \tau$

(we write $a \mapsto \tau_1$, $b \mapsto \tau_2$, $c \mapsto \tau_3$ ...)

The typed λ-calculus

Types (3):

- A Type Judgement stating that a term t has type τ in environments Σ and Γ :

$$\Sigma, \Gamma \vdash t :: \tau$$

- … and a set of type inference rules establishing type judgements.

The typed λ-calculus

- Type Inferences:

$$\frac{}{\Sigma, \Gamma \vdash c_i :: \theta\ (\Sigma\ c_i)} \qquad \frac{}{\Sigma, \Gamma \vdash x_i :: \Gamma\ x_i}$$

$$\frac{\Sigma, \Gamma \vdash E :: \tau \Rightarrow \tau' \quad \Sigma, \Gamma \vdash E' :: \tau}{\Sigma, \Gamma \vdash E\ E' :: \tau'}$$

$$\frac{\Sigma, \{x_i \mapsto \tau\} \uplus \Gamma \vdash E :: \tau'}{\Sigma, \Gamma \vdash \lambda x_i . E :: \tau \Rightarrow \tau'}$$

The typed λ-calculus

- Note that constant symbols where treated slightly different than variable symbols:

  constant symbols may be instantiated (the type variables may be substituted via θ)

  a constant symbol may therefore have different types in a term.

Typed λ-calculus

- We assume Σ =

  {"_+_"↦ nat→nat→nat, "0"↦ nat,"1"↦ nat,"2"↦nat,"3"↦nat,

  "Suc _"↦ nat→nat,

  "_=_"↦ α→α→bool, "True"↦bool),"False" ↦ bool}

Typed λ-calculus

- **Example:** does λx. x + 3 have a type, and which one ?

$$\dfrac{\dfrac{\dfrac{}{\Sigma, \{x \mapsto nat\} \vdash (\_ + \_) :: nat \Rightarrow nat \Rightarrow nat} \quad \dfrac{}{\Sigma, \{x \mapsto nat\} \vdash x :: nat}}{\Sigma, \{x \mapsto nat\} \vdash (\_ + \_)(x) :: nat \Rightarrow nat} \quad \dfrac{}{\Sigma, \{x \mapsto nat\} \vdash 3 :: nat}}{\dfrac{\Sigma, \{x \mapsto nat\} \uplus \{\} \vdash x + 3 :: nat}{\Sigma, \{\} \vdash \lambda x. x + 3 :: nat \Rightarrow nat}}$$

Revisions: Typed λ-calculus

- Examples: Are there variable environments ρ such that the following terms are typable in Σ: (note that we use infix notation: we write "0 + x" instead of "_+_ 0 x")

  - (_+_ 0) = (Suc x)
  - ((x + y) = (y + x)) = False
  - f(_+_ 0) = (λc. g c) x
  - _+_ z (_+_ (Suc 0)) = (0 + f False)
  - a + b = (True = c)

Revisions: β-reduction

- Assume that we want to find typed solutions for ?X, ?Y, ?Z such that the following terms become equivalent modulo α-conversion and β-reduction:
  - ?X a =?= a + ?Y
  - (λc. g c) =?= (λx. ?Y x)
  - (λc. ?X c) a =?= ?Y
  - λa. (λc. X c) a =?= (λx. ?Y)
- Note: Variables like ?X, ?Y, ?Z are called schematic variables; they play a major role in Isabelles Rule-Instantiation Mechanism
- Are the solutions for schematic variables always unique ?

Plan of this Course

- The „λ-calculus“
- α-conversion,β-reduction,ε-reduction
- What is „typed λ-calculus“
- Using typed λ-calculus to represent logical systems
- What is „natural deduction“ ? (from another perspective)

Deduction

- Logic Whirl-Pool of the 20ies (Girard) as response to foundational problems in Mathematics

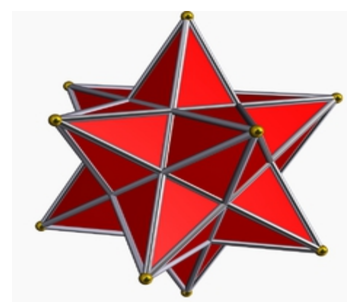
  - growing uneasiness over the question:

    What is a proof ?

    Are there limits of provability ?

Deduction

- Historical context in the 20ies:
  - 1500 false proofs of
    „all parallels do not intersect in infinity“
  - lots of proofs and refutations of
    „all polyhedrons are eularian“ (Lakatosz)



E = F + K - 2 ???

  - Frege`s axiomatic set theory proven inconsistent by Russel
  - Science vs. Marxism debate (Popper)

Deduction

- Historical context in the 20ies:
  - this seemed quite far away from
    Leipnitz vision of

    „Calculemus !“ (We don‘t agree ?
    Let‘s calculate ...)

    of what constitutes, well,

    Science ...

Deduction

- Historical context in the 20ies:
  - attempts to formalize the intuition of „deduction" by Frege, Hilbert, Russel, Lukasiewics, …
  - 2 Calculi presented by Gerhard Gentzen in 1934.
    - „natürliches Schliessen" (natural deduction):
    - „Sequenzkalkül" (sequent calculus)

$$\begin{array}{c}[P]\\ \vdots \\ Q \\ \hline R\end{array}$$

$$\frac{\Gamma \vdash A \vee B \quad \Gamma \cup \{A\} \vdash C \quad \Gamma \cup \{B\} \vdash C}{\Gamma \vdash C}$$

Deduction

- An Inference System (or Logical Calculus) allows to infer formulas from a set of elementary judgements (axioms) and inferred judgements by rules:

$$\frac{A_1 \quad \ldots \quad A_n}{A_{n+1}}$$

"from the assumptions A_1 to A_n, you can infer the conclusion A_{n+1}." A rule with n=0 is an elementary fact. Variables occurring in the formulas A_n can be arbitrarily substituted.

Deduction

- judgements discussed in this course (or elsewhere):

$t : \tau$ “term t has type τ”

$\Gamma \vdash \varphi$ “formula φ is valid under assumptions Γ”

$\vdash$ {P} x:= x+1 {Q} “Hoare Triple”

φ prop “φ is a property”

φ valid “φ is a valid (true) property”

x mortal $\Longrightarrow$ sokrates mortal --- judgements with free variable

etc ...

Natural Deduction

- An Inference System for the equality operator (or “HO Equational Logic”) looks like this:

$$\frac{}{(s = s)prop} \qquad \frac{(s = t)prop}{(t = s)prop} \qquad \frac{(r = s)prop \quad (s = t)prop}{(r = t)prop}$$

$$\frac{(s(x) = t(x))prop}{(s = t)prop}\ where\ x\ is\ fresh \qquad \frac{(s = t)prop \quad (P(s))prop}{(P(t))prop}$$

(where the first rule is an elementary fact).

Natural Deduction

- the same thing presented a bit more neatly (without prop):

$$\frac{}{x = x} \qquad \frac{s = t}{t = s} \qquad \frac{r = s \quad s = t}{r = t}$$

$$\frac{\bigwedge x.\; s\; x = t\; x}{s = t} \qquad \frac{s = t \quad P\; s}{P\; t}$$

(equality on functions as above ("extensional equality") is an HO principle, and it is a classical principle).

Representing logical systems in the typed λ- calculus

- It is straight-forward to use the typed λ-terms as a syntactic means to represent logics; including binding issues related to quantifiers like ∀, ∃, ...
- Example: The Isabelle language „Pure“: It consists of typed λ-terms with constants:
  - foundational types “prop” and “_ => _” (“_ ⇒ _”)
  - the Pure (universal) quantifier

    all :: “(α → Prop) → Prop”

    (“ ⋀x. P x”,“\<And> x. P x”   “!!x. P x”)
  - the Pure implication “A ==> B” (“_ ⟹ _”)
  - the Pure equality “A == B”   “A ≡ B”

„Pure": A (Meta)-Language for Deductive Systems

- Pure is a language to write logical rules.
- Wrt. Isabelle, it is the meta-language, i.e. the built-in formula language.
- Equivalent notations for natural deduction rules:

$A_1 \Longrightarrow (\ldots \Longrightarrow (A_n \Longrightarrow A_{n+1})\ldots),$

$[\![A_1; \ldots; A_n]\!] \Longrightarrow A_{n+1},$

$$\frac{A_1 \quad \ldots \quad A_n}{A_{n+1}}$$

theorem
assumes A_1
and …
and A_n
shows A_{n+1}

„Pure": A (Meta)-Language for Deductive Systems

- Some more complex rules involving the concept of "Discharge" of (formerly hypothetical) assumptions:

$(P \Longrightarrow Q) \Longrightarrow R$:

$$\frac{\begin{matrix}[P] \\ \vdots \\ Q\end{matrix}}{R}$$

theorem
  assumes "P ⟹ Q"
  shows "R"

Propositional Logic as ND calculus

- Some (almost) basic rules in HOL

$$\frac{Q}{\neg\neg Q}$$

$$\frac{\neg\neg Q}{Q}\,\text{notnotE}$$

$$\frac{\begin{array}{c}[A]\\ \vdots \\ B\end{array}}{A \to B}\,\text{impI}$$

$$\frac{A \to B \quad A}{B}\,\text{mp}$$

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

$$\frac{B}{A \vee B}\,\text{disjI2}$$

$$\frac{A \vee B \quad \begin{array}{c}[A]\\ \vdots \\ Q\end{array} \quad \begin{array}{c}[B]\\ \vdots \\ Q\end{array}}{Q}\,\text{disjE}$$

Propositional Logic as ND calculus

- Some (almost) basic rules in HOL

$$\frac{A \wedge B \qquad \begin{matrix}[A, B] \\ \vdots \\ Q\end{matrix}}{Q}\ \text{conjE} \qquad\qquad \frac{A \quad B}{A \wedge B}\ \text{conjI}$$

Key Concepts: Rule-Instances

- A Rule-Instance is a rule where the free variables in its judgements were substituted by a common substitution σ:

$$\frac{A \quad B}{A \wedge B}\,\text{conjI} \quad \xrightarrow{\sigma} \quad \frac{3 < x \quad x \leq y}{3 < x \wedge x \leq y}$$

where σ is $\{A \mapsto 3{<}x,\ B \mapsto x{\leq}y\}$.

Key Concepts: Formal Proofs

- A series of inference rule instances is usually displayed as a Proof Tree (or : Derivation or: Formal Proof)

$$\text{subst}\;\dfrac{\dfrac{\text{sym}\;\dfrac{f(a,b)=a}{a=f(a,b)} \qquad \dfrac{f(a,b)=a \qquad f(f(a,b),b)=c}{f(a,b)=c}\;\text{subst}}{a=c}\;\text{trans} \qquad \dfrac{}{g(a)=g(a)}\;\text{refl}}{g(a)=g(c)}$$

- The hypothetical facts at the leaves are called the assumptions of the proof (here $f(a,b) = a$ and $f(f(a,b),b) = c$).

Key Concepts: Discharge

- A key requisite of ND is the concept of discharge of assumptions allowed by some rules (like impI)

$$\frac{\begin{array}{c}[A]\\ \vdots \\ B\end{array}}{A \to B}$$

$$\dfrac{\dfrac{\dfrac{[f(a,b)=a]}{a = f(a,b)}\,\text{sym} \quad \dfrac{[f(a,b)=a] \quad f(f(a,b),b)=c}{f(a,b)=c}\,\text{subst}}{a=c}\,\text{trans} \qquad \dfrac{}{g(a)=g(a)}\,\text{refl}}{\dfrac{g(a)=g(c)}{f(a,b)=a \to g(a)=g(c)}}\,\text{subst}$$

- The set of assumptions is diminished by the discharged hypothetical facts of the proof (remaining: $f(f(a,b),b) = c$).

Key Concepts: Global Assumptions

- The set of (proof-global) assumptions gives rise to the notation:

$$\{f(a,b) = a, f(f(a,b),b) = c\} \vdash g(a) = g(c)$$

written:

$$A \vdash \phi$$

or when emphasising the global theory (also called: global context):

$$A \vdash_E \phi$$

Sequent-style calculus

- Gentzen introduced and alternative “style” to natural deduction: Sequent style rules.
  - Idea: using the tuples $A \vdash \phi$ as basic judgments of the rules.

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \rightarrow B} \qquad \frac{\Gamma \vdash A \rightarrow B \quad \Gamma \vdash A}{\Gamma \vdash B}$$

Sequent-style calculus

- in contrast to:

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

Sequent-style vs. ND calculus

- Both styles are linked by two transformations called "lifting over assumptions" Lifting over assumptions transforms:

$$\frac{A_1 \quad \ldots \quad A_n}{A_{n+1}}$$

where we consider for the moment $\vdash$ just equivalent to meta implication $\Longrightarrow$

$$\frac{\Gamma \vdash A_1 \quad \ldots \quad \Gamma \vdash A_n}{\Gamma \vdash A_{n+1}}$$

Quantifiers

- When reasoning over logics with quantifiers (such as FOL, set-theory, TLA, …, and of course: HOL), the additional concept of "parameters" of a rule is necessary. We assume that there is an infinite set of variables and that it is always possible to find a "fresh" unused one …

  – Consider:

$$\frac{\forall x. P(x)}{P(t)} \; for\ any\ term\ t$$

$$\frac{\forall x. P(x) \qquad \begin{matrix}[P(y)]_y \\ \vdots \\ Q\end{matrix}}{Q}$$

$$\frac{P(u)}{\forall x. P(x)} \; for\ any\ fresh\ variable\ u$$

$$\frac{P(0) \qquad \begin{matrix}[P(n)]_n \\ \vdots \\ P(Suc\ n)\end{matrix}}{\forall x. P(x)}$$

Quantifiers

- For allI, Isabelle allows certain free variables ?X, ?Y, ?Z that represent „wholes" in a term that can be filled in later by substitution; Coq requires the instantiation when applying the rule.
- Isabelle uses a built-in ("meta")-quantifier $\bigwedge$x. P x already seen on page 13; Coq uses internally a similar concept not explicitly revealed to the user.

Introduction to Isabelle/HOL

Basic HOL Syntax

- HOL (= Higher-Order Logic) goes back to Alonzo Church who invented this in the 30ies …
- "Classical" Logic over the λ-calculus with Curry-style typing (in contrast to Coq)
- Logical type: "bool" injects to "prop". i.e

Trueprop :: "bool ⇒ prop"

is wrapped around any HOL-Term without being printed:

Trueprop A $\Longrightarrow$ Trueprop B is printed: A $\Longrightarrow$ B but A::bool!

Basic HOL Syntax

- Logical connective syntax (Unicode + ASCII):

input:	print:	alt-ascii input
– “_ \<and> _”	“_∧_”	“_ & _”
– “_ \<or> _”	“_ ∨_”	“_ | _
– “_\<longrightarrow>_”	“_ → _”	“_ --> _”
– “_ \<not> _”	“¬_"	“~ _”
– “\<forall> x. P“	“∀x. P”	“! x. P x”
– “\<exists> x. P“	“∃x. P”	“? x. P x”

Basic HOL Rules

- HOL is an equational logic, i.e. a system with the constant "_=_::'a 'a bool" and the rules:

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

$$\frac{\bigwedge x.\ s\ x = t\ x}{s = t}\ \text{ext} \qquad \frac{s = t \quad P\ s}{P\ t}\ \text{subst}$$

Basic HOL Rules

- HOL is an equational logic, i.e. a system with the constant "_=_::'a 'a bool" and the rules:

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

$$\frac{\bigwedge x.\ s\ x = t\ x}{s = t}\ \text{ext} \qquad \frac{s = t \quad P\ s}{P\ t}\ \text{subst}$$

which rule makes HOL „higher-order“ ???

Basic HOL Rules

- Some (almost) basic rules in HOL

$$\frac{A \wedge B \qquad \begin{matrix}[A, B] \\ \vdots \\ Q\end{matrix}}{Q}\,\text{conjE} \qquad\qquad \frac{A \quad B}{A \wedge B}\,\text{conjI}$$

HOL Rules

- The quantifier rules of HOL:

$$\frac{\bigwedge x.\ P\ x}{\forall x. P\ x}\ \text{allI} \qquad \frac{\forall x. P\ x \qquad \begin{matrix}[P\ ?t]\\ \vdots \\ Q\end{matrix}}{Q}\ \text{allE}$$

allE (safe, but incomplete)

again: what makes theses HOL „higher-order" ???

HOL Rules

- The quantifier rules of HOL:

$$\frac{\forall x. P\ x \qquad \begin{array}{c}[P\ ?t; \forall x. P\ x]\\ \vdots \\ Q\end{array}}{Q}$$

allduPE
(unsafe, but
complete)

HOL Rules

- The quantifier rules of HOL:

$$\frac{\forall x. P\ x \qquad \begin{array}{c}[P\ ?t; \forall x. P\ x] \\ \vdots \\ Q\end{array}}{Q}$$

allldupE
(unsafe, but
complete)

HOL Rules

- The quantifier rules of HOL:

$$\frac{P\ ?t}{\exists x. P\ x}\ \text{exI}$$

$$\frac{\exists x.\ P(x) \qquad \begin{array}{c}[P(x)]_x \\ \vdots \\ Q\end{array}}{Q}\ \text{exE}$$

HOL Rules

- From these rules (which were defined actually slightly differently), a large body of other rules can be DERIVED (formally proven, and introduced as new rule in the proof environment).

  Examples: see exercises.

Typed Set-theory in HOL

- The HOL Logic comes immediately with a typed set - theory: The type

  α set $\cong$ $\alpha \Rightarrow$ bool, that's it !

  can be defined isomorphically to its type of characteristic functions !

- THIS GIVES RISE TO A RICH SET THEORY DEVELOPPED IN THE LIBRARY (Set.thy).

Typed Set Theory: Syntax

- Logical connective syntax (Unicode + ASCII):

input:	print:	alt-ascii input
"_ \<in> _"	"_ ∈ _"	"_ : _"
"{_ . _}"	*{x. True ∧ x = x}*	*for example*
"_ \<union> _"	"_∪_"	"_ Un _"
"_ \<inter> _"	"_ ∩_"	"_ Int _"
"_\<subseteq>_"	"_ ⊆ _"	"_ <= _"

Conclusion

- Typed λ-calculus is a rich term language for the representation of logics, logical rules, and logical derivations (proofs)
- On the basis of typed λ-calculus, Higher-order logic (HOL) is fairly easy to represent
- … the differences to first-order logic (FOL) are actually tiny.