Preuves Interactives et Applications

Burkhart Wolff

http://www.lri.fr/~wolff/teach-material/2020-2021/M2-CSMR

Université Paris-Saclay

Foundations: λ-calculus

Motivation: Why ITP* ?

*)Interactive Theorem Proving

- Enormous Versatility of Proof Problems :
  - Mathematics : 4 color theorem, Kepler conjecture,
    Feit-Thompson conjecture. . .
  - Theoretical Informatics :
    - Formal proofs of algorithms
    - Program Language Semantics,
    - Soundness of Type-Systems
  - Engineering (Practical Informatics):
    - Back-end for other provers (certifying proof traces),
    - Discharging Proof Obligations in Program Verification
    - SEL4 (Isabelle/HOL, NICTA), secured micro-kernel for OS
    - CompCert (Coq, Inria), optimising C compiler
- … much stuff in Phd-thesis and the literature …

Plan of this Course: „λ–calculus"

- Untyped λ - Terms
- Conversions and Reductions
- The typed λ–calculus
- Properties
- Encoding Logics in the typed λ–calculus
- What is „natural deduction" ?

Foundations:

Untyped λ-Terms

Background: 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 Terms of the (pure) λ-calculus

- λ-terms T are built (inductively) over:

  - V, a set of "variable symbols"
  - $\lambda\mathrm{V}.\ \mathrm{T}$, a term construction called "λ-abstraction",
  - $\mathrm{T}\ \mathrm{T}$, a term construction called "application"

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

The λ-calculus: Notation

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: Binding

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 $\lambda x.\ x\ x$, x appears "bound". If a variable occurrence in not bound, is called "free".

The λ-calculus : Binding

Example:

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

The free variables can be computed recursively:

- free(x) = {x} for any x ∈
- 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.

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

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

Corrected Example:

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

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 a function by substitution of its 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

All conversions/reductions are congruences, i.e. can be applied to any sub-term.

Substitution and Conversions

Example:

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

Now consider:

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

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

Computations

Example (Church Numerals):

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

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

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

Consider:

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

Substitution and Conversions

Example (Boolean Logics):

TRUE $\equiv \lambda x.\lambda y.x$

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

AND $\equiv \lambda p.\lambda q.p\ q\ p$

OR $\equiv \lambda p.\lambda q.p\ p\ q$

NOT $\equiv \lambda p.$p FALSE TRUE

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

Consider:

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

Substitution and Conversions

Example (Recursive Function):

$$\text{FAC} \equiv \lambda \mathit{fac}.\ \lambda n.\ \mathit{IFTHENELSE}\ (\mathit{ISZERO}\ n)\ (1)\ (\mathit{MULT}\ n\ (\mathit{fac}(\mathit{PRED}\ n)))$$

$$\boldsymbol{Y} \equiv \lambda f.\ (\lambda x.\ f(x\ x))\ (\lambda x.\ f(x\ x))$$

Consider:

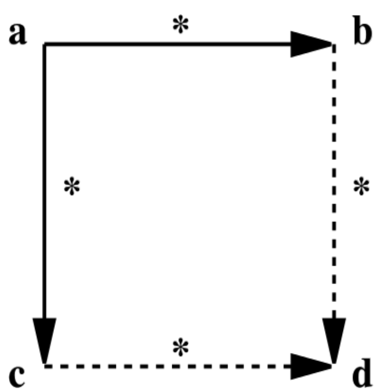
$$(\text{Y FAC})\ 4 \quad \rightarrow_{\beta}^{*} \quad 24$$

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, $\alpha\eta$-conversions)



- the equality on $\lambda-terms$ is undecidable.
- the difference between “Pure” and “Applied” irrelevant

Foundations:
Typed λ-Terms

The typed λ-calculus

Motivation:

- a term - language for representing maths (with quantifiers, integrals, limits and stuff - thus: variables with 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.
- Typed terms were defined inductively.

The typed λ-calculus

(Applied) λ-terms T
are built (inductively) over:

- V, a set of "variable symbols"
- C, a set of "constant symbols"
- $\lambda V.\, T$, a term construction called "λ–abstraction" ,
- $T\ T$, a term construction called "application"

The typed λ-calculus

Types (1):

- We assume a set of type constructors χ with symbols like bool, nat, int, _list, _set, _⇒_, ...
- We assume a set of type variables TV for $\alpha,\beta,\gamma\ldots$
- The set of types τ is inductively defined:

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

The typed λ-calculus

Types (2):

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

The typed λ-calculus

Types (3):

- We assume constant environment which assigns each constant symbol a type:

  $$\Sigma :: C \mapsto \tau$$

- We assume a variable-environment which assigns to each variable symbol a type:

  $$\Gamma :: V \mapsto \tau$$

(we write $\Gamma = \{a \mapsto \tau_1, b \mapsto \tau_2, c \mapsto \tau_3 \ldots\}$)

The typed λ-calculus

Types (4):

- 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 inductive 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

$$\Sigma = \{0 \mapsto nat, 1 \mapsto nat, 2 \mapsto nat, 3 \mapsto nat,$$
$$Suc\ \_ \mapsto nat \Rightarrow nat,\ \ \_+\_ \mapsto nat \Rightarrow nat \Rightarrow nat,$$
$$\_=\_ \mapsto \alpha \Rightarrow \alpha \Rightarrow bool, True \mapsto bool, False \mapsto bool\}$$

Typed λ-calculus

- Example:

  does λx. x + 3 have a type, and which one ?

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

Revisions: Typed λ-calculus

- Examples:

  Are there variable environments ρ such that the following terms are typable in Σ:
  (note the infix notation: we write 0 + x for "_+_" 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 solutions for schematic variables always unique ?

The typed λ-calculus

Theoretical Properties (without proof)

- the congruence

$$t \leftrightarrow_{\alpha\beta\eta} t'$$

  is decidable (reduce to β-normalform, expand to η-longform, rename vars via α in some canonical order)

- Systems like Coq, Isabelle, HOL4 can use (some form of) typed λ-calculi as universal term-representation with binding operators such as ∀, ∃, sums, integrals, ...

The typed λ-calculus

Theoretical Properties (without proof)

- The type inference problem is decidable, i.e. for

$$\Sigma, ? \vdash t :: ??$$

  there is an algorithm that finds solutions for ? and ?? if existing.
- the difference between "Pure" and "Applied" is relevant for typing
- `λf.(λx.f(x x))(λx.f(x x))` is untypable
- Beta-reduction is terminating, i.e. there is always an irreducible t' for any t such that:

$$t \rightarrow_{\beta}^{*} t'$$

Application: Encoding a Simple Logic in typed λ-Terms

Pure in Typed λ-calculus

- We assume for a minimal logic:

$$\Sigma_{\text{Pure}} = \{ \ \_ \Longrightarrow \_ \mapsto \text{prop} \Rightarrow \text{prop} \Rightarrow \text{prop},$$

$$\_ \equiv \_ \quad \mapsto \alpha \Rightarrow \alpha \Rightarrow \text{prop},$$

$$\bigwedge \_.\_ \quad \mapsto (\alpha \Rightarrow \text{prop}) \Rightarrow \text{prop}\}$$

where we will equivalently write

$\bigwedge x.\ P$ for $\bigwedge \_.\_(\lambda x.\ P)$. (Quantifier notation)

HOL in Typed λ-calculus

- We assume for Higher-Order Logic:

$\Sigma_{HOL} = \Sigma_{Pure} \uplus$

$\{$ Trueprop $\mapsto$ bool $\Rightarrow$ prop,

True $\mapsto$ bool, False $\mapsto$ bool,

$\_\wedge\_ \mapsto$ bool$\Rightarrow$bool$\Rightarrow$bool, $\_\vee\_ \mapsto$ bool$\Rightarrow$bool$\Rightarrow$bool,

$\_ \longrightarrow \_ \mapsto$ bool $\Rightarrow$ bool $\Rightarrow$ bool, $\neg\,\_ \mapsto$ bool $\Rightarrow$ bool,

$\_ = \_ \mapsto \alpha \Rightarrow \alpha \Rightarrow$ bool,

$\forall_._ \mapsto (\alpha\Rightarrow$bool$)\Rightarrow$ bool,

$\exists_._ \mapsto (\alpha\Rightarrow$bool$)\Rightarrow$ bool $\}$

Outlook: representing Rules

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

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

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

(Prop is Trueprop and the bar corresponds to A $\Longrightarrow$ B).

Natural Deduction

- With a nicer pretty-printing this looks like this:

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

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

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

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.