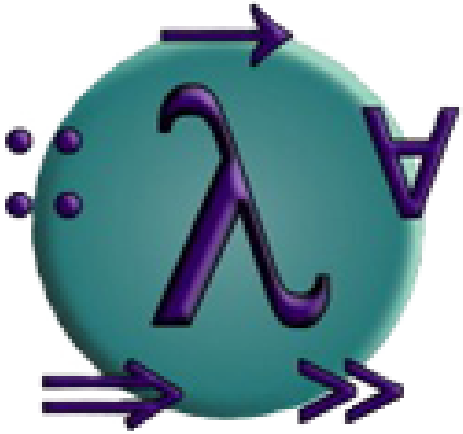
Type Theory

Summer School on Formal Techniques



Dr Stéphane Graham-Lengrand, SRI International

stephane.graham-lengrand@csl.sri.com

In brief

- Type theory can be viewed as an area of Logic...

  ...that is concerned not only with the semantics of formulae (e.g., in $\{\mathsf{true}, \mathsf{false}\}$)

  ...but also with the semantics of proofs

  Semantics of programs is perhaps more natural than semantics of proofs

- Type theory exploits this, based on the *proofs-as-programs* paradigm

  (a.k.a. Curry-Howard-DeBruijn correspondence)

- Type theory is based on types as in a (functional) programming language

  More precisely: based on typed λ-calculus

- Type theory gives the foundation of a wide range of Interactive Theorem Provers:

  Coq, Agda, Lean, Matita, Idris, etc

  To illustrate these lectures today: Coq

- Examples and exercises:

  `http://www.csl.sri.com/users/sgl/ssft/`

Contents

I. The λ-calculus and simple types

The λ-calculus on one slide

Three constructs:

- variables, e.g., x, y, z
- applications, e.g., $t\ u$
- λ-abstractions, e.g., $\lambda x.t$

In other words:

$$t, u, v, \ldots ::= x \mid t\ u \mid \lambda x.t$$

What is $\lambda x.x$? What is $\lambda y.y$?

What is $\lambda x.\lambda y.x$? What is $\lambda x.\lambda y.y\ x$?

β-reduction: $(\lambda x.t)\ u \longrightarrow \{{}^{u}/_{x}\}t$

How does this term reduce?

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

Coq

$$\frac{}{\Delta, x : A \vdash x : A}$$

$$\frac{\Delta \vdash t : A \to B \qquad \Delta \vdash u : A}{\Delta \vdash t\ u : B}$$

$$\frac{\Delta, x : A \vdash t : B}{\Delta \vdash \lambda x.t : A \to B}$$

Where Δ is a typing context, and A and B are types ranging over

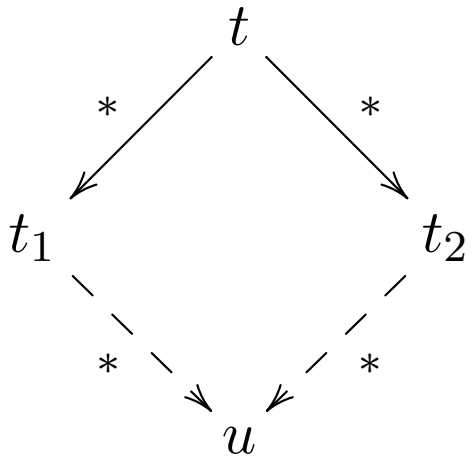
$$A, B, C, \ldots ::= a \mid A \to B$$

(a ranges over base types)

Confluence

Theorem: the relation $\longrightarrow$ is *confluent*, i.e.,

If $t \longrightarrow^* t_1$ and $t \longrightarrow^* t_2$, then there exists u such that $t_1 \longrightarrow^* u$ and $t_2 \longrightarrow^* u$



$\longrightarrow^*$ is reflexive and transitive closure of $\longrightarrow$

$\longleftrightarrow^*$ is reflexive, transitive and symmetric closure of $\longrightarrow$

Proof: not today

Corollary: Irreducible forms are unique, i.e.,

Given a λ-term t, there is at most one irreducible u such that $t \longrightarrow^* u$

Existence of u?

What about $\omega = (\lambda x.x\ x)\ (\lambda x.x\ x)$? What about $(\lambda x.y)\ \omega$?

Motivation for typing

Reason why some terms are not normalising relates to issues such as

whether applying a function to itself ($x\ x$) makes mathematical sense

(Issue very close to whether or not a predicate can apply to itself $P(P)$ or whether a set can belong to itself $y \in y$)

Has to do with controlling the **domain** of function x

In Set theory:

if $f : A \longrightarrow B$ and $x \in A$ then writing $f(x)$ "makes sense" and $f(x) \in B$

But we don't need Set theory for that:

Abstract away from sets to retain only the necessary ingredients for

controlling domains and applications of functions to arguments

$$x \in A \qquad \text{becomes} \qquad x : A$$

This is the notion of *typing*

This is a purely syntactic notion

About typing

Are these λ-terms typable? Coq

$$\lambda x.\lambda y.y\ x$$

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

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

$$(\lambda x.x\ x)\ (\lambda x.x\ x)$$

Remark: If $\Delta \vdash t : A$ then FV(t) is included in the domain of Δ

Reduction preserves typing: If $\Delta \vdash t : A$ and $t \longrightarrow t'$ then $\Delta \vdash t' : A$

Proof: easy induction on the inductive property $t \longrightarrow t'$

Termination: If $\Delta \vdash t : A$ then all reduction paths starting from t are finite

Proof: not today

The λ-calculus

- models the core of functional programming languages
- is used in Higher-Order Logic (HOL) to construct predicates

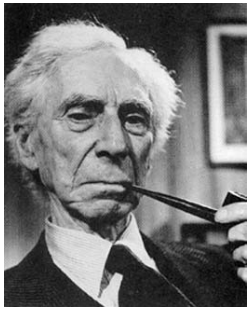
(e.g., $\lambda x^a.\lambda y^a.\forall P^{a\to\mathsf{Prop}}(P\,x \Rightarrow P\,y)$)

In both cases, typing plays an important role.

A slogan (Milner 78): **"Well-typed programs cannot go wrong"**

In Higher-Order Logic, it prevents us from applying a predicate to itself, e.g., $P(P)$.

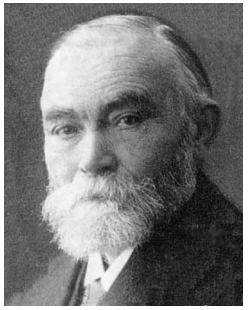
This was (essentially) allowed in Frege's foundation for mathematics, which Russell's paradox showed inconsistent.



Russell to Frege:

"There is just one point where I have encountered a difficulty. You state that a function too, can act as the indeterminate element. This I formerly believed, but now this view seems doubtful to me because of the following contradiction. Let w be the predicate: to be a predicate that cannot be predicated of itself. Can w be predicated of itself? From each answer its opposite follows. Therefore we must conclude that w is not a predicate."

Frege:



"A scientist can hardly meet with anything more undesirable than to have the foundation give way just as the work is finished. In this position I was put by a letter from Mr Bertrand Russell as the work was nearly through the press."

Formalization of the set-theoretic version of the paradox: Coq

Type Theory

Russell and Whitehead fixed the paradox with a Theory of Types,

to restrict the syntax of statements.

Started the reconstruction of mathematics in that framework: Principia Mathematica.

Further developments by Ramsey, then Church.

The resulting system, where typed λ-calculus provides the syntax of statements,

is now known as **Higher-Order Logic**.

A modern twist in Type Theory:

typed λ-calculus also used to represent **proofs**

Connection with proofs

Let's look at fragment of Natural Deduction for implication only:

$$\frac{}{\Gamma, P \vdash P}$$

$$\frac{\Gamma \vdash P \Rightarrow Q \quad \Gamma \vdash P}{\Gamma \vdash Q}$$

$$\frac{\Gamma, P \vdash Q}{\Gamma \vdash P \Rightarrow Q}$$

Now let's look again at the typing rules for λ-calculus:

$$\frac{}{\Delta, x : A \vdash x : A}$$

$$\frac{\Delta \vdash t : A \to B \qquad \Delta \vdash u : A}{\Delta \vdash t\, u : B}$$

$$\frac{\Delta, x : A \vdash t : B}{\Delta \vdash \lambda x.t : A \to B}$$

What can we say?

More precisely

Propositions are Types

Proofs are Programs

Every proof tree can be annotated to be the typing tree of some λ-term

(λ-calculus variables annotate hypotheses, a λ-term annotates the conclusion)

Conversely:

Every typing tree, for some λ-term t, can be turned into a proof tree,

simply by hiding variables and λ-term annotations

It is the **Curry-Howard-DeBruijn isomorphism**

Proving a given statement = finding inhabitant of a given type (proving = programming)

Exercise

Coq

Give a proof of

$\vdash P \Rightarrow P$

What is the λ-term annotating the proof?

Give a proof of

$\vdash P_1 \Rightarrow (P_2 \Rightarrow P_1)$

What is the λ-term annotating the proof?

Give a proof of

$\vdash (P_1 \Rightarrow (P_2 \Rightarrow P_3)) \Rightarrow (P_1 \Rightarrow P_2) \Rightarrow P_1 \Rightarrow P_3$

What is the λ-term annotating the proof?

Give a proof of

$\vdash ((P_1 \Rightarrow P_2) \Rightarrow P_1) \Rightarrow P_1$

What is the truth table for this formula?

II. Intuitionistic logic

The drinker's theorem

"There is always someone such that, if he drinks, everybody drinks"

$$\exists x(\text{DRINKS}(x) \Rightarrow \forall y\ \text{DRINKS}(y))$$

Proof - Informal

Take the first guy you see, call it Bob.

Either Bob does not drink,

in which case he satisfies the predicate “if he drinks, everybody drinks”

. . . or Bob drinks, in which case we have to check that everybody else drinks

If this is the case, then again Bob is the person we are looking for

If we find someone who does not drink, call it Frank,

we change our mind and say that the guy we are looking for is Frank

We can turn this into a formal proof of the formula $\exists x \, (\textsf{DRINKS}(x) \Rightarrow \forall y \, \textsf{DRINKS}(y))$ in predicate logic. . .

. . . using the rule

$$\frac{}{\Gamma \vdash P \vee \neg P} \text{ Law of Excluded Middle}$$

Problem with this

We have proved the theorem

. . . but we are still incapable of identifying the person satisfying the property

(or rather, our choice depends on the context)

In other words, we fail to provide a witness of existence

In other words, the logic we use does not have the witness property

The logic we use lacks a certain dose of constructivism

Lack of witness, another example

Predicate P: assuming $P(0), \neg P(2)$

Can we prove that there is an integer x such that $P(x) \wedge \neg P(x+1)$?

$$P(0), \neg P(2) \vdash \exists x\, (P(x) \wedge \neg P(x+1))$$

Is there an integer n such that we can prove $P(n) \wedge \neg P(n+1)$?

$$P(0), \neg P(2) \vdash P(n) \wedge \neg P(n+1)$$

Concrete example:

Let $u_0 := \sqrt{2}$, $u_{x+1} := u_x^{\sqrt{2}}$, and $P(x)$ be "u_x irrational"

$P(0)$, $P(2)$?

Applying the above: There is x such that $P(x)$ and $\neg P(x+1)$

Therefore: There is an irrational r ($:= u_x$) such that $r^{\sqrt{2}}$ rational

r is either $\sqrt{2}$ or $\sqrt{2}^{\sqrt{2}}$, depending on whether $\sqrt{2}^{\sqrt{2}}$ is rational or not

The mismatch

Remark:

$\vdash P \wedge Q$ if and only if both $\vdash P$ and $\vdash Q$

The object-level $\wedge$ matches the meta-level "and"

$\vdash \forall x\, P[x]$ if and only if for all terms t we have $\vdash P[t]$ (t not necessarily closed)

The object-level $\forall$ matches the meta-level "for all"

Clearly:
If either $\vdash P$ or $\vdash Q$ then $\vdash P \vee Q$
If there is a term t such that $\vdash P[t]$ then $\vdash \exists x\, P[x]$

But	If you have...	...you don't necessarily have
	$\vdash \exists x\, P[x]$	an n such that $\vdash P[n]$
Example	$\vdash \exists x\, (P(x) \vee \neg P(x+1))$	an n such that $\vdash P(n) \vee \neg P(n+1)$
	$\vdash P \vee Q$	either $\vdash P$ or $\vdash Q$
Example	$\vdash P \vee \neg P$	either $\vdash P$ or $\vdash \neg P$ (e.g., P Goedel formula)

For $\vee$ and $\exists$, there is a mismatch between the object-level and the meta-level

The culprit and how to fix the mismatch

In all our examples, mismatch entirely due to:

- the Law of Excluded Middle ($P \vee \neg P$)
- or, equivalently, the Elimination of Double Negation ($(\neg\neg P) \Rightarrow P$).

The fix is easy: **Disallow** those laws

You get what is called **Intuitionistic Logic**(s) -as opposed to Classical logic(s)

Distinction can be done for propositional logic, predicate logic, higher-order logic, etc.

The claim: we recover a full match between object-level and meta-level

- $\vdash P \wedge Q$ if and only if both $\vdash P$ and $\vdash Q$
- $\vdash \forall x\ P[x]$ if and only if for all terms t we have $\vdash P[t]$
- $\vdash P \vee Q$ if and only if either $\vdash P$ or $\vdash Q$
- $\vdash \exists x\ P[x]$ if and only if there is a term t such that $\vdash P[t]$

The above match works in the empty theory, not in any theory!

(imagine the theory $P \vee Q$ or the theory $\exists x\ P$)

III. Computing with intuitionistic proofs

Constructivism

Seeking the witness property is part of a wider approach to mathematics called "constructivism".

Objects that mathematics speak about must be "constructed".

(Typical bad example: the set of all sets, from Russell's paradox)

If we can always speak about an object we have shown to exist,
then while showing its existence we must have constructed it.

Similarly, if we claim one of two things holds, we should be able to compute which one of the two holds.

Constructivism brought about a very computational view of what it is to do mathematics.

A computational interpretation of intuitionistic logic

Following Brouwer–Heyting–Kolmogorov, Kleene proposed that the interpretation $[\![A]\!]$ of a formula A no longer be 0 or 1, but a set of realisers.

Instead of the usual truth tables, the semantics of formulae is governed by the following rules, where $r \in [\![A]\!]$ is written $r \Vdash A$ ("r realises A"):

$r \Vdash P_1 \wedge P_2$	if $r = (r_1, r_2)$ with $r_1 \Vdash P_1$ and $r_2 \Vdash P_2$
$r \Vdash P_1 \vee P_2$	if $r = (i, r')$ with either $i = 1$ and $r' \Vdash P_1$, or $i = 2$ and $r' \Vdash P_2$
	said differently: if $r = \mathrm{inj}_i(r')$ with $r' \Vdash P_i$ for $i = 1$ or $i = 2$
$r \Vdash \exists x P(x)$	if $r = (a, r')$ with a an element of the "model" and $r' \Vdash P(a)$
$r \Vdash P_1 \to P_2$	if r is a computable function such that, whenever $r' \Vdash P_1$, $r(r') \Vdash P_2$
$r \Vdash \forall x P(x)$	if r is a computable function such that, for all elements a of the "model", $r(a) \Vdash P(a)$

Parameterised by a way to interpret atomic formulae

A computational interpretation of intuitionistic logic

$r \Vdash P_1 \wedge P_2$ if $r = (r_1, r_2)$ with $r_1 \Vdash P_1$ and $r_2 \Vdash P_2$

$r \Vdash P_1 \vee P_2$ if $r = (i, r')$ with either $i = 1$ and $r' \Vdash P_1$, or $i = 2$ and $r' \Vdash P_2$

said differently: if $r = \mathsf{inj}_i(r')$ with $r' \Vdash P_i$ for $i = 1$ or $i = 2$

$r \Vdash \exists x P(x)$ if $r = (a, r')$ with a an element of the "model" and $r' \Vdash P(a)$

$r \Vdash P_1 \to P_2$ if r is a computable function such that, whenever $r' \Vdash P_1$, $r(r') \Vdash P_2$

$r \Vdash \forall x P(x)$ if r is a computable function such that,
for all elements a of the "model", $r(a) \Vdash P(a)$

What exactly does r range over?

r ranges over mathematical objects such as pairs, computable functions, etc

r can be implemented as a number Can you say how?

r can be implemented as an untyped λ-term (untyped λ-calculus is Turing-complete)

there comes the Curry-Howard-DeBruijn correspondence

Through the C-H correspondence, intuitionistic proofs provide realisers

The λ-calculus you have seen is for the connective $\Rightarrow$

Theorem : If $\vdash M : P$ then $M \Vdash P$

What about the other connectives?

We can extend the λ-calculus

to account for the introduction and elimination rules of the other connectives

$\wedge$ for product types

$P_1 \wedge P_2$ is a product type "$P_1 * P_2$"

$$\frac{\Gamma \vdash M : P_1 \quad \Gamma \vdash N : P_2}{\Gamma \vdash (M, N) : P_1 \wedge P_2} \qquad \frac{\Gamma \vdash M : P_1 \wedge P_2}{\Gamma \vdash \pi_i(M) : P_i}\, i \in \{1, 2\}$$

Can you give a proof-term for the valid formula $(P_1 \wedge P_2) \Rightarrow (P_2 \wedge P_1)$?

Coq

$\vee$ for sum types

$P_1 \vee P_2$ is a sum type "$P_1 + P_2$"

$$\frac{\Gamma \vdash M : P_i}{\Gamma \vdash \mathsf{inj}_i(M) : P_1 \vee P_2}\, i \in \{1,2\}$$

$$\frac{\Gamma \vdash M : P_1 \vee P_2 \quad \Gamma, \alpha_1 : P_1 \vdash N_1 : Q \quad \Gamma, \alpha_2 : P_2 \vdash N_2 : Q}{\Gamma \vdash \left(\begin{array}{l}\mathsf{match}\ M\ \mathsf{with}\\ \quad \mid \mathsf{inj}_1(\alpha_1).N_1\\ \quad \mid \mathsf{inj}_2(\alpha_2).N_2\end{array}\right) : Q}$$

If your favorite language does not have these constructs, implement $\mathsf{inj}_i(M)$ as (i, M),

and $\left(\begin{array}{l}\mathsf{match}\ M\ \mathsf{with}\\ \quad \mid \mathsf{inj}_1(\alpha_1).N_1\\ \quad \mid \mathsf{inj}_2(\alpha_2).N_2\end{array}\right)$ as $\begin{array}{l}\texttt{if}\ \ \pi_1(M)\ \texttt{=}\ \texttt{1}\\ \texttt{then}\ \ \left\{{}^{\pi_2(M)}\!/_{\alpha_1}\right\} N_1\\ \texttt{else}\ \ \left\{{}^{\pi_2(M)}\!/_{\alpha_2}\right\} N_2\end{array}$

Can you give a proof-term for the valid formula $(P_1 \vee P_2) \Rightarrow (P_2 \vee P_1)$?

Coq

Reductions

$$(\lambda\alpha.M)\ N \longrightarrow \left\{{}^{N}\!/_{\alpha}\right\} M$$

$$\pi_i(M_1, M_2) \longrightarrow M_i$$

$$\left(\begin{array}{l}\mathsf{match}\ \mathsf{inj}_i(M)\ \mathsf{with}\\ \quad \mid \mathsf{inj}_1(\alpha_1).N_1\\ \quad \mid \mathsf{inj}_2(\alpha_2).N_2\end{array}\right) \longrightarrow \left\{{}^{M}\!/_{\alpha_i}\right\} N_i$$

+ some permutation rules such as

$$\left(\begin{array}{l}\mathsf{match}\ M\ \mathsf{with}\\ \quad \mid \mathsf{inj}_1(\alpha_1).N_1\\ \quad \mid \mathsf{inj}_2(\alpha_2).N_2\end{array}\right) N \longrightarrow \begin{array}{l}\mathsf{match}\ M\ \mathsf{with}\\ \quad \mid \mathsf{inj}_1(\alpha_1).(N_1\ N)\\ \quad \mid \mathsf{inj}_2(\alpha_2).(N_2\ N)\end{array}$$

$$\pi_i \left(\begin{array}{l}\mathsf{match}\ M\ \mathsf{with}\\ \quad \mid \mathsf{inj}_1(\alpha_1).N_1\\ \quad \mid \mathsf{inj}_2(\alpha_2).N_2\end{array}\right) \longrightarrow \begin{array}{l}\mathsf{match}\ M\ \mathsf{with}\\ \quad \mid \mathsf{inj}_1(\alpha_1).\pi_i(N_1)\\ \quad \mid \mathsf{inj}_2(\alpha_2).\pi_i(N_2)\end{array}$$

$\ldots$

Recovering the match with the meta-level

Possible to extend the λ-calculus with constructions that capture the rules for $\forall$ and $\exists$

Reduction still preserves typing: If $\Gamma \vdash M : P$ and $M \longrightarrow M'$ then $\Gamma \vdash M' : P$

Through the Curry-Howard-DeBruijn isomorphism,

this is describing a proof transformation process

Termination (proof: not today): We still have termination of typed terms

Corollaries (proof: not today):

- Consistency: There is no closed proof of $\bot$
- Witness property: If $\vdash \exists x\ P[x]$ then there is a term t such that $\vdash P[t]$
- Disjunction property: If $\vdash P_1 \vee P_2$ then either $\vdash P_1$ or $\vdash P_2$

Conclusion: In the empty theory, we recover a full match between object-level and meta-level (in the sense discussed before)

Remark: Law of Excluded Middle would break all of the above approach

In non-empty theories

Axioms labelled by variables but:

These variables have no computational role

The normal forms are not as nice

We lose the Consistency, the Witness property and the Disjunction property

In some theories (e.g., Peano's arithmetic), a computational role can be given to axioms

Theorem holds again, and its corollaries:

Consistency, Witness property, Disjunction property

Programming by proving

In arithmetic, does $\forall x\, \exists y\, (x = 2 \times y \vee x = 2 \times y + 1)$ Coq

have a proof in intuitionistic logic? by induction on x!

What about $\exists y\, (25 = 2 \times y \vee 25 = 2 \times y + 1)$?

What is the witness? How do you compute it?

An intuitionistic proof of

$$\forall x\, \exists y\, (x = 2 \times y \vee x = 2 \times y + 1)$$

is a program that computes the half here by recursion on x!

Its execution mechanism is the proof-transformation process described before

The program is *correct* with respect to the *specification*

$$x = 2 \times y \vee x = 2 \times y + 1$$

IV. Putting it all together: HOL with proof-terms, Dependent types

Using λ-calculus for both propositions and proofs

So far in logic, λ-calculus used

- at the level of propositions in Higher-Order Logic
- at the level of proofs in the Curry-Howard-DeBruijn correspondence

(so far in intuitionistic first-order logic)

Can we have combine the two in one system,

with the λ-calculus operating at both levels?

This means

- equip (the intuitionistic version of) HOL with a notion of proof-terms based on λ-calculus
- equivalently, extend the Curry-Howard-DeBruijn correspondence

  so that the types are the propositions of HOL

This is called System F_ω

Exercises

We can write the following well-formed HOL propositions:

$$\forall x^{\mathsf{Prop}}((\forall y^{\mathsf{Prop}} y) \Rightarrow x)$$

$$\forall x^{\mathsf{Prop}} \forall y^{\mathsf{Prop}} (((x \Rightarrow y) \Rightarrow x) \Rightarrow x)$$

$$\forall x^{a} \forall y^{a} ((\forall z^{a \to \mathsf{Prop}} (z\ x \Rightarrow z\ y)) \Rightarrow (\forall z^{a \to \mathsf{Prop}} (z\ y \Rightarrow z\ x)))$$

Can you find proof-terms for them? Coq

Dependent types

Since in F_ω,

- statements speak about objects described as λ-terms,
- and proofs are described as λ-terms,

could it be possible to have the logic speak about its own proofs?

Yes, by adding to F_ω dependent types.

This gives the **Calculus of Constructions** (CoC).

Example of dependent type: the type of lists of length n, i.e., $\mathsf{list}\ n$

Careful with a logic that speaks about its own proofs:

remember a logic is either inconsistent or cannot prove its own consistency

Inductive types

Many proof assistants (Coq, Matita, Lean, Agda, Epigram, Twelf, Lego, etc) are developed for variants of this logic, with some features removed or added.

Coq, Matita, etc add to the Calculus of Constructions inductive types, which generalise in that logic the algebraic datatypes of ML languages, and are used to represent

- enumerated types, e.g., booleans $\{\mathsf{true}, \mathsf{false}\}$ (different from Prop!)
- tuples, records
- natural numbers
- lists
- trees
- other logical connectives $\wedge$, $\vee$, etc
- existential quantifier
- equality

Coq

V. Special treatment of equality: interpretation of its proofs

Equality

In pure first-order logic, equality can be given by reflexivity + Leibniz

$$\frac{}{t = t} \qquad \frac{}{t = u \Rightarrow P(t) \Rightarrow P(u)}$$

In proof assistants such as Coq, equality itself is an **inductive type**.

More precisely, it is defined as the inductive type

parameterised by a type A and an element $x : A$,

that has 1 constructor, for reflexivity:

$$(\texttt{eq\_refl}_A\ x) \quad : \quad (x =_A x)$$

Does it have the expected properties of equality?

If it has the Leibniz property, then we should be satisfied with this definition, right?

Good news

Elimination principles for inductive types (i.e. pattern-matching)

entail the Leibniz property. Coq

Typing rules for eliminating inductive types are sufficient to get

- Leibniz principle
- disequality of terms of an inductive type with

  different constructors at their head
- constructors of inductive types are injective

For instance, the following axioms of arithmetic are provable

(in-built, no need for axioms)

$$\forall n(0 \neq S\, n)$$

$$\forall nm((S\, n = S\, m) \Rightarrow (n = m))$$

Properties of interest

Let A be a type.

UIP_refl (Unicity of Identity proofs)

$$\forall x^A \forall p^{(x=x)} \quad (p = \texttt{eq\_refl}\ x)$$

Property K

$$\forall x^A \forall P^{(x=x \to \mathsf{Prop})}$$

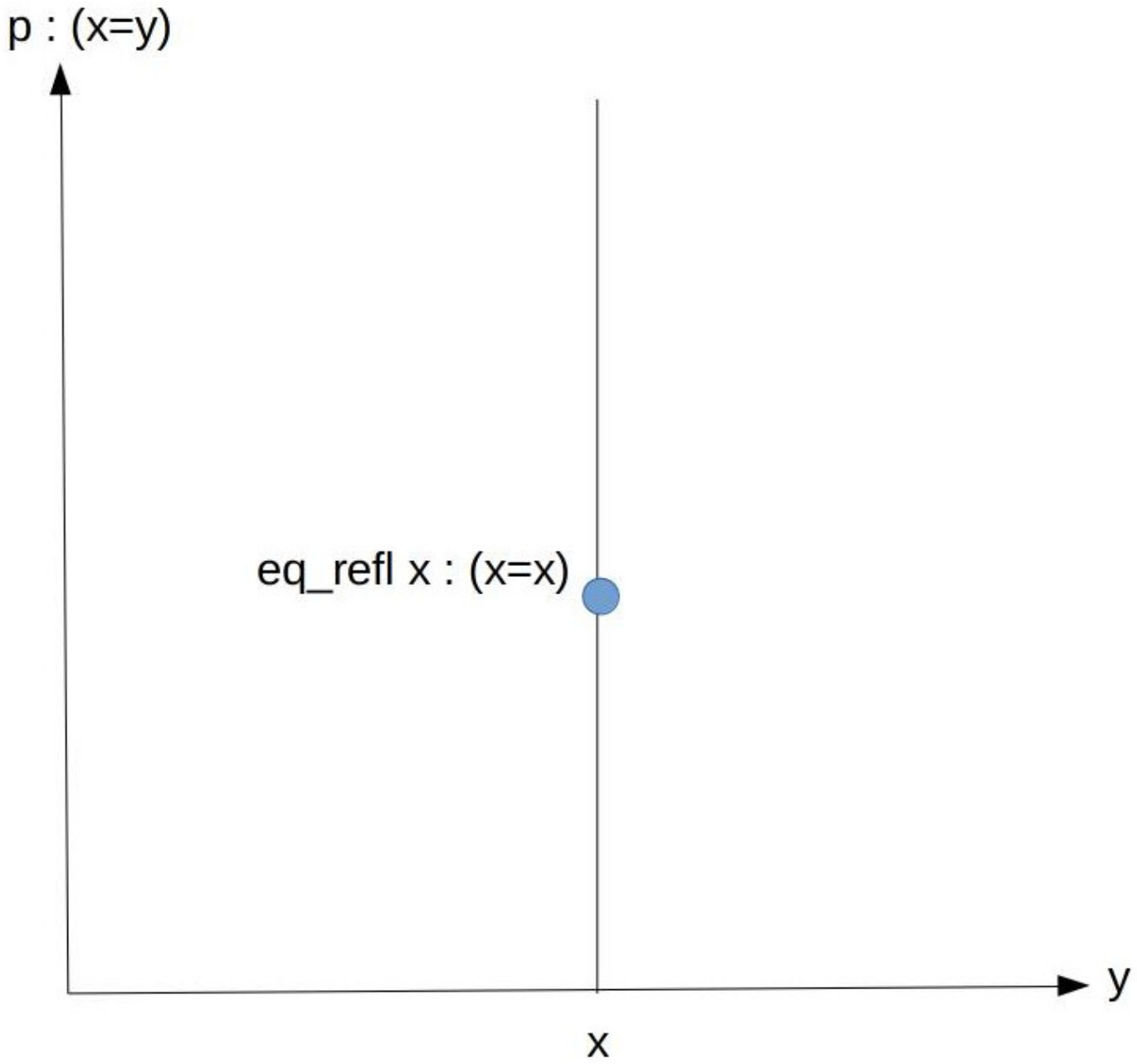
$$P(\texttt{eq\_refl}\ x) \to \forall p^{(x=x)} P(p)$$

Property J (mind the dependent product!)

$$\forall x^A \forall P^{(\{y:A\ \&\ x=y\} \to \mathsf{Prop})}$$

$$P(x, \texttt{eq\_refl}\ x) \to \forall y^A \forall p^{(x=y)} P(y, p)$$

Visualizing the dependent product $\{y : A \,\&\, x = y\}$



Another one for the road: the heterogeneous equality

Binary predicate very similar to equality, but can apply to inhabitants of different types:

If $t : A$ and $u : B$, the term $t =^{\mathsf{JM}}_{A,B} u$ is well-typed.

But of course it only holds if B is A, with a constructor similar to `eq_refl`:

$$(\texttt{JMeq\_refl}_A\ x) \quad : \quad (x =^{\mathsf{JM}}_{A,A} x)$$

But is it equivalent to regular equality? In other words, can we prove:

$$\forall x^A \forall y^A\ (x =^{\mathsf{JM}}_{A,A} y\ \rightarrow\ x =_A y)$$

Surprise

Only J is provable.

If the others are not provable, there must be a counter-model.

However, given a function deciding equality in A, then K, UIP_refl, etc do hold (Hedberg'98).

Building a model

Usually in type theory, every type A is interpreted as a set $[\![A]\!]$,

and every inhabitant $t : A$ is interpreted as an element $[\![t]\!] \in [\![A]\!]$

Usually in logic, equality has a special treatment

that declares $t =_A u$ as satisfied by the model iff $[\![t]\!]$ is the same element as $[\![u]\!]$.

Assume we apply this to $=$ (and $=^{\mathsf{JM}}$), naturally defining the interpretation $[\![t =_A u]\!]$

as a singleton set $\{\bullet\}$ if $[\![t]\!]$ is $[\![u]\!]$

and as the empty set $\emptyset$ if not.

This will not give a counter-model.

So we need to be more imaginative. What do we know about equality?

- We have reflexivity: $(\mathtt{eq\_refl}\ t) : (t = t)$
- We have symmetry:

  we can reverse $p : (t = u)$ into $p^{-1} : (u = t)$
- We have transitivity:

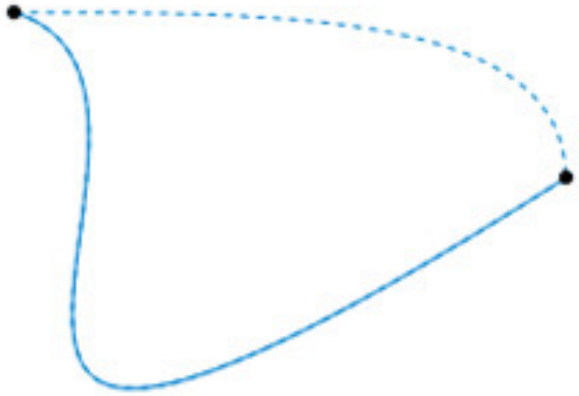
  we can compose $p : (t_1 = t_2)$ and $p' : (t_2 = t_3)$ into $p \circ p' : (t_1 = t_3)$

This gives rise to the Groupoid model of Type Theory (Hofmann-Streicher'96)

A concrete instance

An idea is to interpret types as topological spaces,

and interpret the proofs of $t = u$ as the (continuous) paths from $[\![t]\!]$ to $[\![u]\!]$



- We have the trivial path from $[\![t]\!]$ to $[\![t]\!]$
- We can reverse a path p from $[\![t]\!]$ to $[\![u]\!]$ into a path from path p^{-1} from $[\![u]\!]$ to $[\![t]\!]$
- We can compose a path p from $[\![t_1]\!]$ to $[\![t_2]\!]$ and a path p' from $[\![t_2]\!]$ to $[\![t_3]\!]$ into a path $p \circ p'$ from $[\![t_1]\!]$ to $[\![t_3]\!]$

This is **Homotopy Type Theory** (HoTT)

A few points about Homotopy Type Theory

- There is an equality between t and u if they belong to the same connected component in the interpretation of their type

- There can be different proofs of an equality $t = u$, interpreted as different paths

- We can formalise a notion of equality between two proofs of equality π and π'
  as a homotopy, i.e. a continuous deformation of one path into the other
  (this is how paths of a topological space form a topological space)

- We can state the equality between two proofs of equality between proofs of equality, and so on and so forth...

Conclusion

- Homotopy theorists like to impose the univalence axiom,

  which entails that isomorphic types are equal

  (this does not come for free in standard type theory)

- A characterisation of usual notions comes out of the following hierarchy:
  - "Propositions" are types that are either empty or entirely connected,

    with the types of equalities entirely connected,

    the types of equalities between equalities entirely connected, etc
  - "Sets" are types whose elements may be equal "in at most one way",

    i.e., equalities between its elements are propositions
  - etc

Main thing to take away

Via the Curry-Howard-DeBruijn correspondence

Programming = Proving

Proving proposition A = Inhabiting type A

VI. Appendix

Naïve set theory

Syntax: Empty term signature; Predicate signature: $\in$ and $=$, arity 2

In naive set theory, we have, for every formula A with free variables $\{x_1, \ldots, x_n, y\}$, an axiom

$$\forall x_1 \ldots \forall x_n \exists c \forall y \, (y \in c \Leftrightarrow A(x_1, \ldots, x_n, y))$$

Informally:

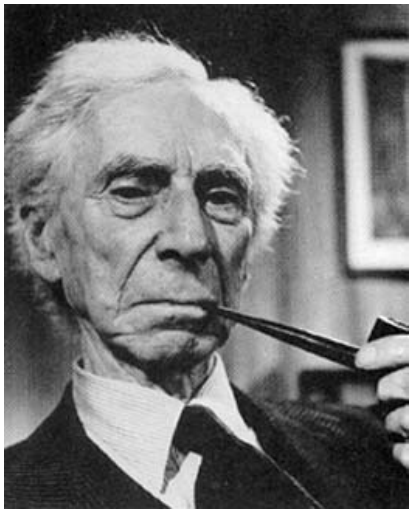
existence of the set $\{y \mid A(y)\}$, i.e., the set of all elements y satisfying $A(y)$

First instantiation: in particular we have

$$\exists r \forall y \, (y \in r \Leftrightarrow \top)$$

i.e., $\exists r \forall y \, (y \in r)$ (there is a set of all sets)

Russell's paradox (1902) expressed in naive set theory



Second instantiation: in particular we have axiom R:

$$\exists r \forall y \, (y \in r \Leftrightarrow \neg y \in y)$$

What about $r \in r$? or is it $\neg r \in r$?

Clearly, $R \vdash \exists r (r \in r \Leftrightarrow \neg r \in r)$

Is this problematic?

Yes, since:

Lemma: Given a theory $\mathcal{T}$, if $\mathcal{T} \vdash (A \Leftrightarrow \neg A)$ then $\mathcal{T} \vdash \bot$

Can you prove it?

Frege's system

In fact, Frege's Begriffschrift (1879) considers functions as **more primitive** than sets

They are part of his logic (rather than the purpose of an axiomatic theory)

Frege distinguishes *objects* (which include natural numbers) and *functions*

- Variables $x, y, z, \ldots$ represent objects, while variables $f, g, h, \ldots$ represent functions
- (The representation of) a function can be applied to (the representation of) an object, so $f(x)$ represents an object
- Both kinds of variables can be quantified over: $\forall x \ldots$ and $\forall f \ldots$
- *predicates*[1] are those functions mapping objects to either 0 or 1
- *formulae* are the various representations of 0 and 1
- Given a formula $A[x]$, Frege allows the representation, in his syntax, of the function that maps an object c to the object represented by $A[c]$. In modern notations, such a function is denoted by the λ-term $\lambda x.A[x]$
- His system allows to prove the *simplification property*: $\forall y\ (((\lambda x.A[x])(y)) \Leftrightarrow A[y])$

[1] Frege called them "concepts"

Frege's system

Notice that we are already outside the syntax of first-order logic, since

- there are 2 kinds of variables and terms (instead of 1): for objects and for functions
- formulae are particular object terms & $\lambda x.A[x]$ builds a function term from a formula
- moreover $\lambda x.A[x]$ and $\lambda y.A[y]$ are identified as the same term

$\Longrightarrow$ notion of binder in the syntax of terms

But so far so good, no one has found any contradiction in this system

But in 1893, Frege adds a tool for creating objects from functions

(to speak about equality of functions)

In the case of a predicate F,

ϵF denotes a particular object called the *extension of concept* F,

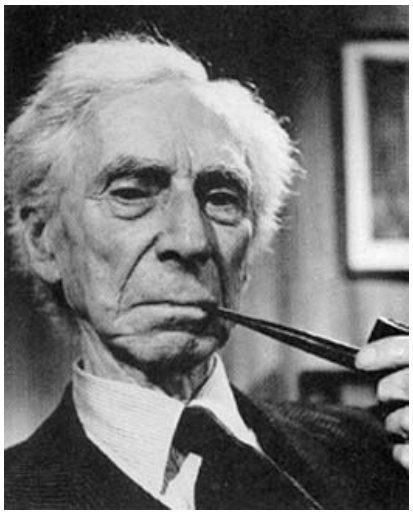
satisfying *"Basic Law V"* (Axiom 5 of Frege's system):

$$\forall F \forall G \left((\epsilon F = \epsilon G) \Leftrightarrow \forall x \left(Fx \Leftrightarrow Gx \right) \right)$$

Russell's paradox (1902) expressed in Frege's system

Russell expresses his paradox both in terms of set theory and in Frege's logical system

(with functions)



Let P be the following predicate term:

$$\lambda x.\exists F\,(x = \epsilon F \wedge \neg F(x))$$

Clearly, $P(\epsilon P)$ means

$$(\lambda x.\exists F\,(x = \epsilon F \wedge \neg F(x)))(\epsilon P)$$

and by the simplification property this is equivalent to

$$\exists F\,(\epsilon P = \epsilon F \wedge \neg F(\epsilon P))$$

Then by Basic Law V we know that

$$\epsilon P = \epsilon F \text{ is equivalent to } \forall x\,(Px \Leftrightarrow Fx)$$

so the above is equivalent to

$$\neg P(\epsilon P)$$

In Frege's system, there is a formula $P(\epsilon P)$ equivalent to its negation

$\Longrightarrow$ in Frege's system **there is a proof of** $\bot$

No set involved, but of course in substance ϵP is "the set of all objects satisfying P"