Type Systems and Logics

Maria João Frade
Departamento de Informática
Universidade do Minho
2007

Program Semantics, Verification, and Construction

MAP-i, Braga 2007

Program Semantics, **Verification**, and Construction

Type Systems and Logics

Maria João Frade

Departamento de Informática
Universidade do Minho

MAP-i, Braga 2007

Part II - Program Verification

- Proof assistants based on type theory
  - Type Systems and Logics
    - Pure Type Systems
    - The Lambda Cube
    - The Logic Cube
  - Extensions of Pure Type Systems
    - Sigma Types
    - Inductive Types
    - The Calculus of Inductive Constructions
    - Introduction to the Coq proof assistant
- The Coq proof assistant
- Axiomatic semantics of imperative programs: Hoare Logic
- Tool support for the specification, verification, and certification of programs

Bibliography

- Henk Barendregt. Lambda calculi with types. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, Handbook of Logic in Computer Science, volume 2, pages 117–309. Oxford Science Publications, 1992.

- Henk Barendregt and Herman Geuvers. Proof-assistants using dependent type systems. In John Alan Robinson and Andrei Voronkov, editors, Handbook of Automated Reasoning, pages 1149–1238. Elsevier and MIT Press, 2001.

- Gilles Barthe and Thierry Coquand. An introduction to dependent type theory. In Gilles Barthe, Peter Dybjer, Luís Pinto, and João Saraiva, editors, APPSEM, volume 2395 of Lecture Notes in Computer Science, pages 1–41. Springer, 2000.

- Yves Bertot and Pierre Castéran. Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions, volume XXV of Texts in Theoretical Com- puter Science. An EATCS Series. Springer Verlag, 2004.

- http://coq.inria.fr/. Documentation of the Coq proof assistant (version 8.1).

Proof Checking

- Proof checking consists of the automated verification of mathematical theories.
  - First one formalizes within a given logic the underlying primitive notions, the definitions, the axioms and the proofs;
  - and then the definitions are checked for their well-formedness and the proofs for their correctness.

  In this way mathematics is represented on a computer and also a hight degree of reliability is obtained.

- Once the theory is formalized, its correctness can be verified by the **proof-checker** (which is a small program).

- To help in the formalization process there exists an interactive **proof-development system**.

- Proof-checker and proof-development systems are usually combined in what is called a **proof-assistant**.

Proof-assistants

In a proof-assistant, after formalizing the primitive notions of the theory (under study), the user develops the proofs interactively by means of (proof) tactics, and when a proof is finished a "proof-term" is created. This proof-term closely corresponds to a standard mathematical proof (in natural deduction style).

Machine assisted theorem proving:

- helps to deal with large problems;
- prevents us from overseeing details;
- does the bookkeeping of the proofs.

Proof-assistants based on type theory present a general specification language to define mathematical notions and formulas. Moreover, it allows to construct algorithms and proofs as first class citizens.

Proof checking mathematical statements

- Mathematics is usually presented in an informal but precise way.

> In situation Γ we have A.
> Proof. p. QED

- In Logic Γ, A become formal objects and proofs can be formalized as a derivation tree (following some precisely given set of rules).

> $\Gamma \vdash_L A$
> Proof. p. QED

Types in logic

- The connection of type theory to logic is via the proposition-as-types principle that establishes a precise relation between intuitionistic logic and computation.
- Intuitionistic logic is based on the notion of proof – a proposition is true when we can provide a constructive proof of it. On this basis:
  - a proposition A can be seen as a type (the type of its proofs);
  - and a proof of A as an object of type A.

Hence: A is **provable** $\Leftrightarrow$ A is **inhabited**

Therefore, the formalization of mathematics in type theory becomes

$\Gamma \vdash_T p : A$ which is equivalent to $\text{Type}_\Gamma(p) = A$

So, proof checking boils down to **type checking**.

Type-theoretic notions for proof-checking

In the practice of an interactive proof assistant based on type theory, the user types in tactics, guiding the proof development system to construct a proof-term. At the end, this term is type checked and the type is compared with the original goal.

In connection to proof checking there are some decidability problems:

Type Checking Problem (TCP) $\Gamma \vdash_T M : A \;\; ?$

Type Synthesis Problem (TSP) $\Gamma \vdash_T M : ?$

Type Inhabitation Problem (TIP) $\Gamma \vdash_T \; ? : A$

TIP is usually undecidable for type theories of interest.

TCP and TSP are decidable for a large class of interesting type theories.

The reliability of machine checked proofs

- **Why would one believe a system that says it has verified a proof ?**

  The proof checker should be a very small program that can be verified by hand, giving the highest possible reliability to the proof checker.

- **de Bruijn criterion**

  A proof assistant satisfies the de Bruijn criterion if it generates proof-objects (of some form) that can be checked by an 'easy' algorithm.

Proof-objects may be large but they are self-evident. This means that a small program can verify them. The program just follows whether locally the correct steps are being made.

Type-theoretic approach to interactive theorem proving

provability of formula A	$\Longleftrightarrow$	inhabitation of type A
proof checking	$\Longleftrightarrow$	type checking
interactive theorem proving	$\Longleftrightarrow$	interactive construction of a term of a given type

So, decidability of type checking is at the core of the type-theoretic approach to theorem proving.

Examples of proof assistants based on type theory

The first systems of proof checking (type checking) based on the propositions-as-types principle were the systems of the **AUTOMATH** project.

Modern proof assistants aggregate to the proof checker a proof-development system for helping the user to develop the proofs interactively.

We can mention as examples of proof assistants, the systems:

- **Coq** , based on the Calculus of Inductive Constructions
- **Lego** , based on the Extended Calculus of Constructions
- **Alf** and **Agda** , based on Martin-Löf 's type theory
- **Nuprl** , based on extensional Martin-Löf 's type theory

Encoding of logic in type theory

Direct encoding

- Each logical construction have a counterpart in the type theory.
- Theorem proving consists of the (interactive) construction of a **proof-term, which can be easily checked independently**.
- Examples: **Coq**, **Lego**, **Agda**.

Shallow encoding (Logical Frameworks)

- The type theory is used as a logical framework, a meta system for encoding a specific logic one wants to work with.
- The enconding of a logic L is done by choosing an appropriate context Γ_{L}, in which the language of L and the proof rules as declared.
- Usually, the proof-assistants based on this kind of enconding **do not produce standard proof-objects**, just **proof-scripts**.
- Examples:
  - **HOL**, based on the Church's simple type theory. This is a classical higher-order logic.
  - **Isabelle**, based on intuitionistic simple type theory (used as the meta logic). Various logics (FOL, HOL, sequent calculi,...) are described.

Type Systems and Logics

Intuitionistic (constructive) logic

- A proof of $A \supset B$ is a method that transforms a proof of A into a proof of B.
- A proof of $A \wedge B$ is a pair (p, q) such that p is a proof of A and q is a proof of B.
- A proof of $A \vee B$ is a pair (b, p) where b is either 0 or 1 and, if $b{=}0$ then p is a proof of A; if $b{=}1$ then p is a proof of B.
- There is no proof of $\bot$, the false proposition.
- Negation $\neg A$ is defined as $A \supset \bot$.
- A proof of $\forall x \in X.\, P\, x$ is a method p that transforms every element $a \in X$ into a proof of $P a$.
- A proof of $\exists x \in X.\, P\, x$ is a pair (a, p) such that $a \in X$ and p is a proof of $P a$.

Propositions as types

A proposition A is interpreted as the collection of its proofs, represented by $[A]$.

So, according to the intuitionistic interpretation of the logical connectives one has

$$
\begin{array}{rcl}
[A \supset B] & = & [A] \to [B] \\
[A \wedge B] & = & [A] \times [B] \\
[A \vee B] & = & [A] \uplus [B] \\
[\bot] & = & \emptyset \\
[\forall x \in X.\, Px] & = & \Pi\, x{:}X.\, [Px] \\
[\exists x \in X.\, Px] & = & \Sigma\, x{:}X.\, [Px]
\end{array}
$$

where

$$
\begin{array}{rcl}
P \to Q & = & \{f \mid \forall p{:}P.\, f(p) : Q\} \\
P \times Q & = & \{(p,q) \mid p{:}P \ \text{ and } \ q{:}Q\} \\
P \uplus Q & = & \{(0,p) \mid p{:}P\} \bigcup \{(1,q) \mid q{:}Q\} \\
\Pi\, x{:}A.\, Bx & = & \left\{f : (A \to \bigcup_{x:A} Bx) \mid \forall a{:}A.\, (fa : Ba)\right\} \\
\Sigma\, x{:}A.\, Bx & = & \{(a,p) \mid a{:}A \ \text{ and } \ p{:}(Ba)\}
\end{array}
$$

Example

Let X be a set and R be a binary relation on X. Now, consider the following lemma:

If $\forall x, y \in X.\ Rxy \supset \neg Ryx$ then $\forall x \in X.\ \neg Rxx$.

How can this be formalized ?

We have two universes Set and Prop

- a term X of type Set is a type that represents a domain of the logic;
- a term A : Prop is a type that represents a proposition of the logic;
- a predicate on X is represented by a term $P : X \to$ Prop

  $t : X$ satisfies the predicate P iff the type $(P\,t)$ is inhabited
  (i.e., there is a proof-term of type $(P\,t)$)

- a binary relation over X is represented by a term $R : X \to X \to$ Prop.

Example (cont.)

The collection of binary relations over X is represented as $X \to X \to \mathsf{Prop}$.

So, to represent the notion of (polymorphic) binary relation one has to abstract over the domains.

Let us define $\quad \mathsf{Rel} := \lambda X\!:\!\mathsf{Set}.\, X \to X \to \mathsf{Prop}$

Definitions are formal constructions in type theory with a computational rule associated, called **δ-reduction** by which definitions are unfolded.

$$\mathsf{D} \to_{\delta} M \qquad \text{if } \ D := M$$

Anti-symmetry and irreflexivity can also be define as follows

$$\begin{array}{lcl} \mathsf{AntiSym} & := & \lambda X\!:\!\mathsf{Set}.\,\lambda R\!:\!(\mathsf{Rel} X).\,\forall x,y\!:\!X.\,Rxy \supset (Ryx \supset \bot) \\ \mathsf{Irrefl} & := & \lambda X\!:\!\mathsf{Set}.\lambda R\!:\!(\mathsf{Rel} X).\,\forall x\!:\!X.\,Rxx \supset \bot \end{array}$$

Note that $\neg A$ is defined as $A \supset \bot$ where $\bot$ is the empty type (the false proposition).

Example (cont.)

By δ and β-reductions we find that for X : Set and $Q : X \to X \to \mathsf{Prop}$

$$\begin{array}{lcl} (\mathsf{Rel}\, X) & =_{\delta\beta} & X \to X \to \mathsf{Prop} \\ (\mathsf{AntiSym}\, XQ) & =_{\delta\beta} & \forall x,y\!:\!X.\,Qxy \supset (Qyx \supset \bot) \\ (\mathsf{Irrefl}\, XQ) & =_{\delta\beta} & \forall x\!:\!X.\,Qxx \supset \bot \end{array}$$

Here we have a **dependent type**, i.e., a type of functions f where the range-set depends on the input value.

The type of this kind of functions is $f : \Pi x : A.\, B$, the product of a family $\{Bx\}_{x:A}$ of types.

Example (cont.)

The type of dependent functions is $f : \Pi x : A.\, B$, the product of a family $\{Bx\}_{x:A}$ of types.

Intuitively $$\Pi x : A.\, Bx \;=\; \left\{ f : (A \to \bigcup_{x:A} Bx) \mid \forall a : A.\, (fa : Ba) \right\}$$

The typing rules associated are

$$(\text{abstraction}) \quad \frac{\Gamma, x : A \;\vdash\; b : B}{\Gamma \;\vdash\; \lambda x : A.b : (\Pi x : A.\, B)}$$

$$(\text{application}) \quad \frac{\Gamma \;\vdash\; f : (\Pi x : A.\, B) \quad \Gamma \;\vdash\; a : A}{\Gamma \;\vdash\; f\, a : B[x := a]}$$

Note substitution $[x := a]$ in the type of the application.

So, the formula $\forall x : X.\, Qxx \supset \bot$ is translated as the dependent function type

$$\Pi x : X.\, Qxx \to \bot$$

Example (cont.)

Therefore,

$$\begin{array}{lcl} (\mathsf{AntiSym}\; X Q) & = & \Pi x, y : X.\, Qxy \to (Qyx \to \bot) \\ (\mathsf{Irrefl}\; X Q) & = & \Pi x : X.\, Qxx \to \bot \end{array}$$

To prove that anti-symmetry implies irreflexivity for binary relations we have to find a proof-term of type

$$\Pi X : \mathsf{Set}.\, \Pi R : (\mathsf{Rel} X).\, (\mathsf{AntiSym}\; X R) \to (\mathsf{Irrefl}\; X R)$$

the following term is of this type

$$\lambda X : \mathsf{Set}.\, \lambda R : (\mathsf{Rel} X).\, \lambda h : (\mathsf{AntiSym}\; X R).\, \lambda x : X.\, \lambda q : (Rxx).\, hxxqq$$

The verification of this claim is performed by the type-checking algorithm.

Simply-typed λ-calculus is not enough

Simply-typed λ-calculus has not enough expressive power to encode the kind of logic used in the previous example.

There are several type systems embedding some of the features described in our example. For example:

- **System F** – features polymorphism
- **λP** – features dependent types
- **System Fω**– features higher-order polymorphism
- **CC** – features dependent types and higher-order polymorphism

There is a general class of typed λ-calculi were all these systems can be described – the **Pure Type Systems**.

Pure Type Systems

- Pure Type Systems (PTS) provide a general description for a large class of typed λ-calculi.
- PTS make it possible to derive lot of meta theoretic properties in a generic way.
- In PTS we only have one type constructor (Π) and one computation rule (β). (Therefore the name "pure").
- PTS were originally introduced (albeit in a different from) by S. Berardi and J. Terlouw as a generalization of Barendregt's λ-cube, which itself provides a fine-grained analysis of the Calculus of Constructions.

Pure Type Systems

PTS are formal systems for deriving judgments of the form

$$\Gamma \vdash M : A$$

context
list of variable declarations

pseudo-terms

M is of type A relative to a typing of the free variables of M and A (which are declared in Γ)

Syntax

PTS have a single category of expressions, which are called **pseudo-terms**.

The definitions of pseudo-terms is parameterized by a set $\mathcal{V}$ of **variables** and a set S of **sorts** (constants that denote the universes of the type system).

Definition

The set $\mathcal{T}$ of **pseudo-terms** are defined by the abstract syntax

$$\mathcal{T} \ ::= \ \mathcal{S} \mid \mathcal{V} \mid \mathcal{T}\,\mathcal{T} \mid \lambda \mathcal{V}{:}\mathcal{T}.\mathcal{T} \mid \Pi \mathcal{V}{:}\mathcal{T}.\mathcal{T}$$

Both Π and λ bind variables.
We have the usual notation for free variables and bound variables.

Definitions

Pseudo-terms inherit much of the standard definitions and notations of λ-calculi.

- FV(M) denotes the set of free variables of the pseudo-term M .
- We write $A \to B$ instead of $\Pi\, x : A.\, B$ whenever $x \notin$ FV(B).
- $M\,[x := N\,]$ denotes the substitution of N for all the free occurrences of x in M .
- We identify pseudo-terms that are equal up to a renaming of bound variables (**α-conversion**).
- We assume the standard variable convention, so all bound variables are chosen to be different from free variables.

Definitions

- **β-reduction** is defined as the compatible closure of the rule

$$(\lambda\, x : A.M)\, N \quad \to_\beta \quad M[x := N]$$

$\twoheadrightarrow_\beta$ is the reflexive-transitive closure of $\to_\beta$

$=_\beta$ is the reflexive-symmetric-transitive closure of $\to_\beta$

- Application associates to the left, abstraction to the right and application binds more tightly than abstraction.
- We let x, y, z , ... range over $\mathcal{V}$ and s, s', ... range over S

Salient Features of PTS

- PTS describe λ-calculi à la Church (λ-abstractions carry the domain of bound variables).
- PTS are minimal (just Π type construction and β reduction rule), which imposes strict limitations on their applicability.
- PTS model dependent types. Type constructor Π captures in the type theory the set-theoretic notion of generic or dependent function space.

Dependent types

In the type theory one can define for every set A and A-indexed family of sets $(B_a)_{x \in A}$ a new set $\Pi_{x \in A} B_x$ called dependent function space.

Elements of $\Pi_{x \in A} B_x$ are functions with domain A and such that $f(a) \in B_a$ for every $a \in A$.

Π-construction of PTS works in the same way:

$\Pi\, x\!:\!A.\, B(x)$ is the type of terms F such that, for every $a : A$, $F\, a : B(a)$

Specifications

The typing system of PTS is parameterized by a triple $(\mathcal{S}, \mathcal{A}, \mathcal{R})$ where

$\mathcal{S}$ is the set of universes of the type system;
$\mathcal{A}$ determine the typing relation between universes;
$\mathcal{R}$ determine which dependent function types may be found and where they live.

Definition

A PTS-**specification** is a triple $(\mathcal{S}, \mathcal{A}, \mathcal{R})$ where

- $\mathcal{S}$ is a set of **sorts**
- $\mathcal{A} \subseteq \mathcal{S} \times \mathcal{S}$ is a set of **axioms**
- $\mathcal{R} \subseteq \mathcal{S} \times \mathcal{S} \times \mathcal{S}$ is a set of **rules**

We use $(s1,s2)$ to denote rules of the form $(s1,s2,s2)$.

Every specification $\mathbf{S}$ induces a PTS $\lambda\mathbf{S}$.

Contexts and Judgments

- The set $\mathcal{G}$ of **contexts** is given by the abstract syntax $\mathcal{G} ::= \langle\rangle \mid \mathcal{G}, \mathcal{V} : \mathcal{T}$
  - We let $\subseteq$ denote context inclusion
  - The domain of a context is defined by the clause

    $$\mathrm{dom}(x_1 : A_1, ..., x_n : A_n) = \{x_1, ..., x_n\}$$

  - We let Γ, Δ range over $\mathcal{G}$
- A **judgment** is a triple of the form $\Gamma \vdash A : B$ where $A, B \in \mathcal{T}$ and $\Gamma \in \mathcal{G}$.
- A judgment is **derivable** if it can be inferred from the typing rules of the next slide.
  - If $\Gamma \vdash A : B$ then Γ, A and B are legal.
  - If $\Gamma \vdash A : s$ for $s \in \mathcal{S}$, we say that A is a type.

Typing rules for PTS

(axiom)	$\langle\rangle \vdash s_1 : s_2$	if $(s_1, s_2) \in \mathcal{A}$
(start)	$\dfrac{\Gamma \vdash A : s}{\Gamma, x{:}A \vdash x : A}$	if $x \notin \mathsf{dom}(\Gamma)$
(weakening)	$\dfrac{\Gamma \vdash A : B \quad \Gamma \vdash C : s}{\Gamma, x{:}C \vdash A : B}$	if $x \notin \mathsf{dom}(\Gamma)$
(product)	$\dfrac{\Gamma \vdash A : s_1 \quad \Gamma, x{:}A \vdash B : s_2}{\Gamma \vdash (\Pi x{:}A.\, B) : s_3}$	if $(s_1, s_2, s_3) \in \mathcal{R}$
(application)	$\dfrac{\Gamma \vdash F : (\Pi x{:}A.\, B) \quad \Gamma \vdash a : A}{\Gamma \vdash F\, a : B[x := a]}$	
(abstraction)	$\dfrac{\Gamma, x{:}A \vdash b : B \quad \Gamma \vdash (\Pi x{:}A.\, B) : s}{\Gamma \vdash \lambda x{:}A.b : (\Pi x{:}A.\, B)}$	
(conversion)	$\dfrac{\Gamma \vdash A : B \quad \Gamma \vdash B' : s}{\Gamma \vdash A : B'}$	if $B =_\beta B'$

Typing rules for PTS

(axiom) $\langle\rangle \vdash s_1 : s_2$ if $(s_1, s_2) \in \mathcal{A}$

It embeds the relation $\mathcal{A}$ into the type system.

Typing rules for PTS

$$\text{(start)} \quad \frac{\Gamma \vdash A : s}{\Gamma, x\!:\!A \vdash x : A} \quad \text{if } x \notin \mathsf{dom}(\Gamma)$$

$$\text{(weakening)} \quad \frac{\Gamma \vdash A : B \quad \Gamma \vdash C : s}{\Gamma, x\!:\!C \vdash A : B} \quad \text{if } x \notin \mathsf{dom}(\Gamma)$$

It allows the introduction of variables in a context.

Typing rules for PTS

$$\text{(product)} \quad \frac{\Gamma \vdash A : s_1 \quad \Gamma, x\!:\!A \vdash B : s_2}{\Gamma \vdash (\Pi\, x\!:\!A.\, B) : s_3} \quad \text{if } (s_1, s_2, s_3) \in \mathcal{R}$$

It allows for dependent function types to be formed, provided they match the rule in $\mathcal{R}$.

Typing rules for PTS

$$(\text{application}) \quad \frac{\Gamma \vdash F : (\Pi\, x{:}A.\, B) \quad \Gamma \vdash a : A}{\Gamma \vdash F\, a : B[x := a]}$$

It allows to form applications.

Note substitution $[x := a]$ in the type of the application, in order to accommodate type dependencies.

Typing rules for PTS

$$(\text{abstraction}) \quad \frac{\Gamma, x{:}A \vdash b : B \quad \Gamma \vdash (\Pi\, x{:}A.\, B) : s}{\Gamma \vdash \lambda\, x{:}A.b : (\Pi\, x{:}A.\, B)}$$

It allows to build λ-abstractions.

Note that the side condition requires that the dependent function type is well formed.

Typing rules for PTS

$$(\text{conversion}) \quad \frac{\Gamma \vdash A : B \quad \Gamma \vdash B' : s}{\Gamma \vdash A : B'} \quad \text{if } B =_\beta B'$$

It ensures that convertible types (i.e. types that are β-equal) have the same inhabitants.

This rule is crucial for higher-order type theories, because types are λ-terms and can be reduced, and for dependent type theories, because terms may occur in types.

Examples of PTS

Non-dependent type systems (i.e. an expression $M : A$ with $A : *$ cannot appear as a subexpression of $B : *$)

λ→, the simply typed λ-calculus.

$\lambda\!\rightarrow$	$\mathcal{S}$	$=$	$*, \square$
	$\mathcal{A}$	$=$	$(* : \square)$
	$\mathcal{R}$	$=$	$(*, *)$

λ2 is the PTS counterpart of Girard's System F.

$\lambda 2$	$\mathcal{S}$	$=$	$*, \square$
	$\mathcal{A}$	$=$	$(* : \square)$
	$\mathcal{R}$	$=$	$(*, *),\ (\square, *)$

λω is the PTS counterpart of Girard's System Fω.

$\lambda\omega$	$\mathcal{S}$	$=$	$*, \square$
	$\mathcal{A}$	$=$	$(* : \square)$
	$\mathcal{R}$	$=$	$(*, *),\ (\square, *),\ (\square, \square)$

In logical terms, these non-dependent systems correspond to propositional logics.

More examples of non-dependent PTS

λU^-, Girard's System U^-

λU^-	$\mathcal{S}$	$=$	$*,\ \square,\ \triangle$
	$\mathcal{A}$	$=$	$(* : \square),\ (\square : \triangle)$
	$\mathcal{R}$	$=$	$(*,*),\ (\square,*),\ (\square,\square),\ (\triangle,\square)$

λU , System U

λU	$\mathcal{S}$	$=$	$*,\ \square,\ \triangle$
	$\mathcal{A}$	$=$	$(* : \square),\ (\square : \triangle)$
	$\mathcal{R}$	$=$	$(*,*),\ (\square,*),\ (\square,\square),\ (\triangle,*),\ (\triangle,\square)$

The System $\lambda *$

$\lambda *$	$\mathcal{S}$	$=$	$*$
	$\mathcal{A}$	$=$	$(* : *)$
	$\mathcal{R}$	$=$	$(*,*)$

λU^-, λU and $\lambda *$ are **inconsistent** in the sense that there exists a pseudo-term M such that the judgment $A : * \vdash M : A$ is derivable.

Examples of dependent PTS

It is possible to type expressions $B : *$ which contain as subexpression $M : A : *$.

λP is the PTS counterpart of the Logical Frameworks due to Harper et al.

λP	$\mathcal{S}$	$=$	$*,\ \square$
	$\mathcal{A}$	$=$	$(* : \square)$
	$\mathcal{R}$	$=$	$(*,*),\ (*,\square)$

λP2 is the PTS counterpart of Longo and Moggi's system also named λP2.

$\lambda P2$	$\mathcal{S}$	$=$	$*,\ \square$
	$\mathcal{A}$	$=$	$(* : \square)$
	$\mathcal{R}$	$=$	$(*,*),\ (\square,*),\ (*,\square)$

λC (also known as **λPω**) is the PTS counterpart of Coquand and Huet's Calculus of Constructions.

λC	$\mathcal{S}$	$=$	$*,\ \square$
	$\mathcal{A}$	$=$	$(* : \square)$
	$\mathcal{R}$	$=$	$(*,*),\ (\square,*),\ (*,\square),\ (\square,\square)$

In logical terms, these dependent systems correspond to predicate logics.

Another example of dependent PTS

λCω is an extension of the Calculus os Constructions.

λC^ω	$\mathcal{S} = *, \Box_i \quad , i \in \mathrm{N}$ $\mathcal{A} = (* : \Box_0), (\Box_i : \Box_{i+1}) \quad , i \in \mathrm{N}$ $\mathcal{R} = (*, *), (\Box_i, *), (*, \Box_i), (\Box_i, \Box_j, \Box_{\mathsf{max}(i,j)}) \quad , i, j \in \mathrm{N}$

Properties of PTS

Substitution property

If $\Gamma, x : B, \Delta \vdash M : A$ and $\Gamma \vdash N : B$, then $\Gamma, \Delta[x := N] \vdash M[x := N] : A[x := N]$.

Correctness of types

If $\Gamma \vdash A : B$, then either $B \in \mathcal{S}$ or $\exists s \in \mathcal{S}. \Gamma \vdash B : s$.

Thinning

If $\Gamma \vdash A : B$ is legal and $\Gamma \subseteq \Delta$, then $\Delta \vdash A : B$.

Strengthening

If $\Gamma_1, x : A, \Gamma_2 \vdash M : B$ and $x \notin \mathsf{FV}(\Gamma_2) \cup \mathsf{FV}(M) \cup \mathsf{FV}(B)$, then $\Gamma_1, \Gamma_2 \vdash M : B$.

Properties of PTS (cont.)

Confluence

Let $M, N \in \mathcal{T}$. If $M =_\beta N$, then $M \twoheadrightarrow_\beta P$ and $N \twoheadrightarrow_\beta P$ for some $P \in \mathcal{T}$.

Subject Reduction

If $\Gamma \vdash M : A$ and $M \twoheadrightarrow_\beta N$, then $\Gamma \vdash N : A$.

Uniqueness of types

If $\Gamma \vdash M : A$ and $\Gamma \vdash M : B$, then $A =_\beta B$.

Holds if $\mathcal{A} \subseteq \mathcal{S} \times \mathcal{S}$ and $\mathcal{R} \subseteq (\mathcal{S} \times \mathcal{S}) \times \mathcal{S}$ are functions.

Type Checking, Type Inference and Type Inhabitation

Problems one would like to have an algorithm for:

Type Checking Problem (TCP)	$\Gamma \vdash_T M : A$?
Type Synthesis Problem (TSP)	$\Gamma \vdash_T M : ?$
Type Inhabitation Problem (TIP)	$\Gamma \vdash_T ? : A$

In practice, TCP and TSP are very much related:

When checking whether $M\,N : C$ one has to infer a type for N, say A, and a type for M, say D, and then to check whether for some B, $D =_\beta \Pi x{:}A.\,B$ with $B[x := N] =_\beta C$.

- For λ→ all these problems are decidable.
- TIP is undecidable for extensions of λ→ (as it corresponds to the provability in some logic).

Strong Normalization and Decidability of Type Checking

Normalization and Type Checking are intimately connected due to conversion rule.

Strong Normalization (SN)

If $\Gamma \vdash M : A$ then all β-reductions from M terminate.

SN holds for some PTS (e.g., all subsystems of **λC**) and for some not (e.g., λU⁻, λ∗).

A PTS is (weakly or strongly) **normalizing** if all its legal terms are (weakly or strongly) normalizing.

Decidability of Type Checking

In a PTS that is (weakly or strongly) normalizing and with S finite, the problems of type checking and type synthesis are decidable.

Barendregt's λ-Cube

Barendregt's λ-Cube was proposed as a fine-grained analysis of the Calculus of Constructions.

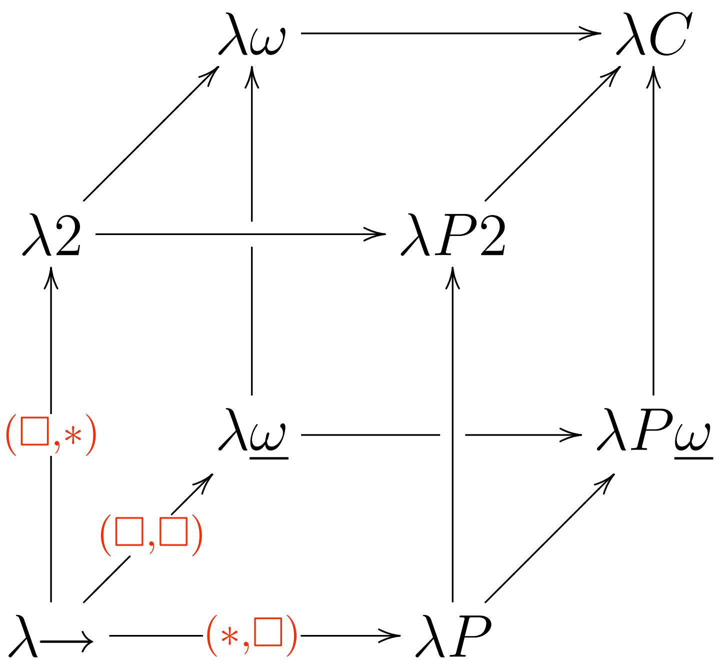
The λ-Cube

The cube of typed lambda calculi consists of eight PTS all of them having $\mathcal{S} = \{*, \square\}$, and $\mathcal{A} = \{* : \square\}$ and the rules for each system as follows:

System	$\mathcal{R}$			
$\lambda\!\rightarrow$	$(*, *)$			
$\lambda 2$	$(*, *)$	$(\square, *)$		
λP	$(*, *)$		$(*, \square)$	
$\lambda\underline{\omega}$	$(*, *)$			$(\square, \square)$
$\lambda\omega$	$(*, *)$	$(\square, *)$		$(\square, \square)$
$\lambda P2$	$(*, *)$	$(\square, *)$	$(*, \square)$	
$\lambda P\underline{\omega}$	$(*, *)$		$(*, \square)$	$(\square, \square)$
λC	$(*, *)$	$(\square, *)$	$(*, \square)$	$(\square, \square)$

The λ-Cube

Note that arrows denote inclusion of one system in another.



Dependencies

Let us call **"types"** to the pseudo-terms of type $*$ and **"kinds"** to the pseudo-terms of type $\Box$.

term : type : kind

- **($*$, $*$)** Terms depending on terms. **(functions)**

$$\vdash\ (\lambda x\!:\!\sigma.\, x) : \sigma \rightarrow \sigma$$

- **($\Box$, $*$)** Terms depending on types. **(polymorphism)**

$$\vdash\ (\lambda \alpha\!:\!*.\lambda x\!:\!\alpha.\, x) : \Pi \alpha\!:\!*.\, \alpha \rightarrow \alpha$$

- **($*$, $\Box$)** Types depending on terms. **(dependent functions)**

$$A : *, P : A \rightarrow * \ \vdash\ (\lambda a\!:\!A.\lambda x\!:\!Pa.\, x) : \Pi a\!:\!A.\, Pa \rightarrow Pa$$

- **($\Box$, $\Box$)** Types depending on types. **(constructors of a kind)**

$$\vdash\ (\lambda \alpha\!:\!*.\alpha \rightarrow \alpha) : \ * \rightarrow *$$

Logics as PTS

Other examples of PTS were given by Berardi who defined logical systems as PTS.

Eight systems of intuitionistic logic will be introduced that correspond in some sense to the systems in the λ-cube. Four systems of proposition logic and four systems of many-sorted predicate logic.

λPROP	proposition logic
λPROP2	second-order proposition logic
λPROP$\underline{\omega}$	weakly higher-order proposition logic
λPROPω	higher-order proposition logic
λPRED	predicate logic
λPRED2	second-order predicate logic
λPRED$\underline{\omega}$	weakly higher-order predicate logic
λPREDω	higher-order predicate logic

Salient features

- All the systems are minimal logics in the sense that the only logical operators are $\supset$ and $\forall$.
- However, for the second and higher-order systems the operators $\neg$, $\wedge$, $\vee$ and $\exists$, as well as Leibeniz's equality are all definable.
- Classical versions of the logics in the upper-plane (of the cube) are obtained easily (by adding the axiom $\forall\alpha.\neg\neg\alpha \to \alpha$).

Berardi's Logic Cube

The Logic Cube

The cube of logical typed lambda calculi consists of the following eight PTS. Each of them has

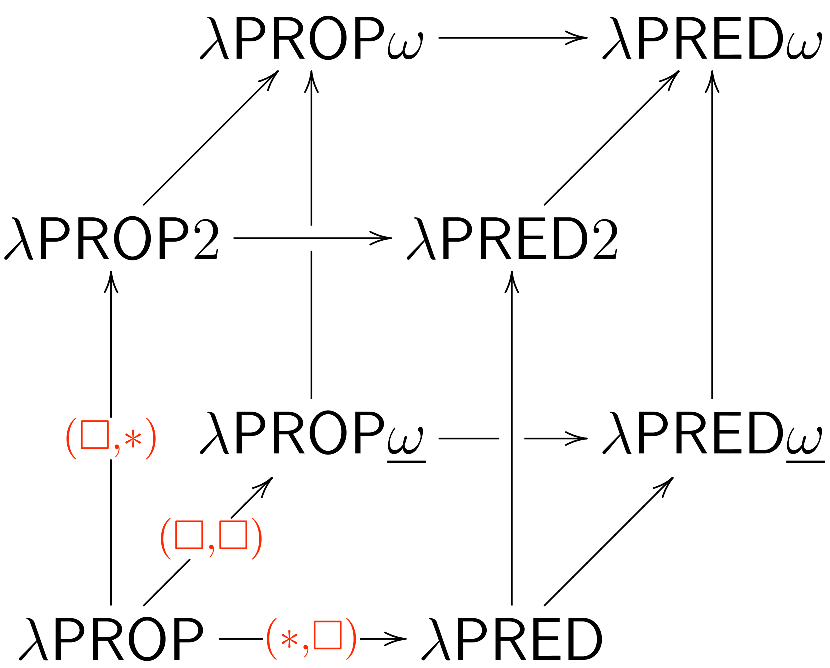
$$\mathcal{S} = \mathsf{Prop},\ \mathsf{Set},\ \mathsf{Type}^p,\ \mathsf{Type}^s$$
$$\mathcal{A} = (\mathsf{Prop} : \mathsf{Type}^p),\ (\mathsf{Set} : \mathsf{Type}^s)$$

and the rules for each of the systems are

System	$\mathcal{R}$			
λPROP				
	(Prop, Prop)			
λPROP2				
	(Prop, Prop)		(Type^p, Prop)	
λPROP$\underline{\omega}$			(Type^p, Type^p)	
	(Prop, Prop)			
λPROPω			(Type^p, Type^p)	
	(Prop, Prop)		(Type^p, Prop)	
λPRED	(Set, Set)	(Set, Type^p)		
	(Prop, Prop)	(Set, Prop)		
λPRED2	(Set, Set)	(Set, Type^p)		
	(Prop, Prop)	(Set, Prop)	(Type^p, Prop)	
λPRED$\underline{\omega}$	(Set, Set)	(Set, Type^p)	(Type^p, Set)	(Type^p, Type^p)
	(Prop, Prop)	(Set, Prop)		
λPREDω	(Set, Set)	(Set, Type^p)	(Type^p, Set)	(Type^p, Type^p)
	(Prop, Prop)	(Set, Prop)	(Type^p, Prop)	

Set is the class of sets and Prop is the class of propositions.

The Logic Cube



Dependencies

The sorts Set and Type^p form the universes of domains.

- $A_1 \to \ldots \to A_n \to \alpha$ with α : Set are functional types.
- $A_1 \to \ldots \to A_n \to \text{Prop}$ are predicate types.

The sort Type^s allows the introduction of variables of type Set.

- (Prop, Prop) allows the formation of implication of two formulae

$$\phi : \text{Prop}, \psi : \text{Prop} \vdash \phi \to \psi : \text{Prop}$$

- (Set, Prop) allows quantification over sets

$$A : \text{Set}, \phi : \text{Prop} \vdash \underbrace{(\Pi x : A.\, \phi)}_{\forall x:A.\phi} : \text{Prop}$$

Dependencies (cont.)

- (Set, Type^p) allows the formation of first-order predicates

$$A : \text{Set} \vdash A \to \text{Prop} : \text{Type}^p$$

hence $$A : \text{Set}, P : A \to \text{Prop}, x : A \vdash Px : \text{Prop}$$

P is a predicate over a set A.

- (Type^p , Prop) allows quantification over predicate types

$$A : \text{Set} \vdash \underbrace{(\Pi P : A \to \text{Prop}.\, \Pi x : A.\, Px \to Px)}_{\forall P{:}A\to\text{Prop}.\,\forall x{:}A.\,Px\to Px} : \text{Prop}$$

Dependencies (cont.)

- (Set, Set) allows function types

$$A : \mathsf{Set}, B : \mathsf{Set} \vdash A \to B : \mathsf{Set}$$

$$\frac{\dfrac{\vdots}{A : \mathsf{Set}, B : \mathsf{Set} \vdash A : \mathsf{Set}} \quad \dfrac{\vdots}{A : \mathsf{Set}, B : \mathsf{Set}, x : A \vdash B : \mathsf{Set}}}{A : \mathsf{Set}, B : \mathsf{Set} \vdash \underbrace{A \to B}_{\Pi x : A.B} : \mathsf{Set}} \ (\mathsf{Set}, \mathsf{Set})$$

- (Type^p , Type^p) allows higher order types

$$A : \mathsf{Set} \vdash (\Pi P : A \to \mathsf{Prop}.\, \mathsf{Prop}) : \mathsf{Type}^p$$

$$\frac{\dfrac{\vdots}{A : \mathsf{Set} \vdash A \to \mathsf{Prop} : \mathsf{Type}^p} \quad \dfrac{\vdots}{A : \mathsf{Set}, P : A \to \mathsf{Prop} \vdash \mathsf{Prop} : \mathsf{Type}^p}}{A : \mathsf{Set} \vdash (\Pi P : A \to \mathsf{Prop}.\, \mathsf{Prop}) : \mathsf{Type}^p} \ (\mathsf{Type}^p, \mathsf{Type}^p)$$

Example of a derivation tree

$$\frac{\dfrac{\vdash \mathsf{Set} : \mathsf{Type}^s}{A : \mathsf{Set} \vdash A : \mathsf{Set}} \quad \dfrac{\dfrac{\vdash \mathsf{Prop} : \mathsf{Type}^p \quad \vdash \mathsf{Set} : \mathsf{Type}^s}{A : \mathsf{Set} \vdash \mathsf{Prop} : \mathsf{Type}^p} \quad \dfrac{\vdash \mathsf{Set} : \mathsf{Type}^s}{A : \mathsf{Set} \vdash A : \mathsf{Set}}}{A : \mathsf{Set}, y : A \vdash \mathsf{Prop} : \mathsf{Type}^p}}{A : \mathsf{Set} \vdash A \to \mathsf{Prop} : \mathsf{Type}^p} \ (\mathsf{Set}, \mathsf{Type}^p) \qquad (2.1)$$

$$\frac{\dfrac{\vdots}{A : \mathsf{Set}, P : A \to \mathsf{Prop}, x : A \vdash P : A \to \mathsf{Prop}} \quad \dfrac{\vdots}{A : \mathsf{Set}, P : A \to \mathsf{Prop}, x : A \vdash x : A}}{A : \mathsf{Set}, P : A \to \mathsf{Prop}, x : A \vdash Px : \mathsf{Prop}} \qquad (2.2)$$

$$\frac{(2.2) \quad (2.2)}{A : \mathsf{Set}, P : A \to \mathsf{Prop}, x : A, q : Px \vdash Px : \mathsf{Prop}} \qquad (2.3)$$

$$\frac{(2.1) \quad \dfrac{\dfrac{\vdots}{A : \mathsf{Set}, P : A \to \mathsf{Prop}, x : A \vdash A : \mathsf{Set}} \quad \dfrac{(2.2) \quad (2.3)}{A : \mathsf{Set}, P : A \to \mathsf{Prop}, x : A \vdash Px \to Px : \mathsf{Prop}} \ \begin{matrix}(\mathsf{Prop}, \mathsf{Prop})\\(\mathsf{Set}, \mathsf{Prop})\end{matrix}}{A : \mathsf{Set}, P : A \to \mathsf{Prop} \vdash (\Pi x : A.\, Px \to Px) : \mathsf{Prop}}}{A : \mathsf{Set} \vdash (\Pi P : A \to \mathsf{Prop}.\, \Pi x : A.\, Px \to Px) : \mathsf{Prop}} \ (\mathsf{Type}^p, \mathsf{Prop})$$

Second-order definability of the logical operations

Despite the logical construction directly encoded in PTS are implication and universal quantification, it is a well known fact in that the upper-plane of the cube the logic connectives $\wedge$, $\vee$, $\bot$, $\neg$ and $\exists$ are definable in terms of $\supset$ and $\forall$.

- For A, B : Prop define

$$\begin{array}{rcl} \bot & \equiv & \Pi\alpha:\text{Prop}.\,\alpha \\ \neg A & \equiv & A\to\bot \\ A\wedge B & \equiv & \Pi\alpha:\text{Prop}.\,(A\to B\to\alpha)\to\alpha \\ A\vee B & \equiv & \Pi\alpha:\text{Prop}.\,(A\to\alpha)\to(B\to\alpha)\to\alpha \end{array}$$

- For A : Prop and X : Set define

$$\exists x:X.A \;\equiv\; \Pi\alpha:\text{Prop}.\,(\Pi x:X.\,A\to\alpha)\to\alpha$$

- For X : Set and $x, y : X$ define the equality predicate $=_L$ called Leibniz equality.

$$(x =_L y) \;\equiv\; \Pi P:X\to\text{Prop}.\,Px\to Py$$

Examples

It is not difficult to check that the intuitionistic elimination and introduction rules for the logic connectives ($\wedge$, $\vee$, $\bot$, $\neg$ and $\exists$) are sound.

Remember $\quad A\wedge B \;\equiv\; \Pi\alpha:\text{Prop}.\,(A\to B\to\alpha)\to\alpha$

Elimination rules

$$\frac{A\wedge B}{A}\;(\wedge\text{E}_1) \qquad A:\text{Prop}, B:\text{Prop}, p:A\wedge B \;\vdash\; pA(\lambda x:A.\,\lambda y:B.\,x):A$$

$$\frac{A\wedge B}{B}\;(\wedge\text{E}_2) \qquad A:\text{Prop}, B:\text{Prop}, p:A\wedge B \;\vdash\; pB(\lambda x:A.\,\lambda y:B.\,y):B$$

Introduction rule

$$\frac{A \quad B}{A\wedge B}\;(\wedge\text{I}) \qquad A:\text{Prop}, B:\text{Prop}, a:A, b:B \;\vdash\; (\lambda\alpha:\text{Prop}.\,\lambda p:(A\to B\to\alpha).\,pab):A\wedge B$$

Examples (cont.)

Note that $A : \mathsf{Prop}, B : \mathsf{Prop} \vdash A \wedge B : \mathsf{Prop}$ can be derived in λPROP2,

but the term $\mathsf{AND} \equiv \lambda A{:}\mathsf{Prop}.\, \lambda B{:}\mathsf{Prop}.\, A \wedge B$ cannot.

One has to be in λPROPω to derive $\vdash \mathsf{AND} : \mathsf{Prop} \to \mathsf{Prop} \to \mathsf{Prop}$

ex falso sequitur quodlibet

$$\frac{\bot}{A}\ (\mathit{ex\ falso}) \qquad\qquad A : \mathsf{Prop}, p : \Pi\alpha{:}\mathsf{Prop}.\alpha \vdash pA : A$$

Examples (cont.)

Let us now prove reflexivity and symmetry for the Leibniz equality. Remember that for X : Set, x, y : X

$$(x =_L y) \equiv \Pi P{:}X \to \mathsf{Prop}.\, Px \to Py$$

Reflexivity $X : \mathsf{Set}, x : X \vdash \underbrace{(\lambda P{:}X \to \mathsf{Prop}.\, \lambda q{:}Px.\, q)}_{\mathsf{w}} : (x =_L x)$

Symmetry

Let $\Gamma \equiv X : \mathsf{Set}, x : X, y : X, t : (x =_L y)$

$$\dfrac{\dfrac{\dfrac{\Gamma \vdash t : (x =_L y) \qquad \Gamma \vdash (\lambda z{:}X.\, z =_L x) : X \to \mathsf{Prop}}{\Gamma \vdash t(\lambda z{:}X.\, z =_L x) : (\lambda z{:}X.\, z =_L x)x \to (\lambda z{:}X.\, z =_L x)y}\ \vdots\ (=_\beta)}{\Gamma \vdash t(\lambda z{:}X.\, z =_L x) : (x =_L x) \to (y =_L x)} \qquad \Gamma \vdash \mathsf{w} : (x =_L x)}{\Gamma \vdash t(\lambda z{:}X.\, z =_L x)\mathsf{w} : (y =_L x)}$$

So,

$$X : \mathsf{Set}, x : X, y : X, t : (x =_L y) \vdash t(\lambda z{:}X.\, z =_L x)(\lambda P{:}X \to \mathsf{Prop}.\, \lambda q{:}Px.\, q) : (y =_L x)$$

Exercices

- Check the soundness of intuitionistic elimination and introduction rules for the other logic connectives.
- Check that the Leibniz equality is transitive.