Program Semantics, **Verification**, and Construction

Type Systems and Logics

Maria João Frade

Departamento de Informática
Universidade do Minho

MAP-i, Porto 2008

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.

$$\boxed{\begin{array}{l}\text{In situation } \Gamma \text{ we have } A.\\ \text{Proof. } p. \text{ QED}\end{array}}$$

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

$$\boxed{\begin{array}{l}\Gamma \ \vdash_L \ A\\ \text{Proof. } p. \text{ QED}\end{array}}$$

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

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

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) $\Gamma \vdash_T M : A \quad ?$

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 $\perp$, the false proposition.
- Negation $\neg A$ is defined as $A \supset \perp$.
- A proof of $\forall x \in X.\, P\, x$ is a method p that transforms every element $a \in X$ into a proof of Pa.
- 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 Pa.

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

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

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:

$$\boxed{\text{If} \quad \forall x, y \in X.\; Rxy \supset \neg\, Ryx \quad \text{then} \quad \forall x \in X.\; \neg\, Rxx\,.}$$

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 $X \to X \to \text{Prop}$.

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

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

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

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

Anti-symmetry and irreflexivity can also be define as follows

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

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

Example (cont.)

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

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

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.