Program Semantics, **Verification**, and Construction

Beyond Pure Type Systems

Maria João Frade

Departamento de Informática
Universidade do Minho

MAP-i, Braga 2007

Part II - Program Verification

- **Proof assistants based on type theory**
  - **Type System and Logics**
    - Pure Type Systems
    - The Lambda Cube
    - The Logic Cube
  - **Extensions of Pure Type Systems**
    - Sigma Systems
    - 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).

Extensions of Pure Type Systems

Extensions of PTS

PTS are minimal languages and lack type-theoretical constructs to carry out practical programming. Several features are not present in PTS. For example:

- It is possible to define data types but one does not get induction over these data types for free. (It is possible to define functions by recursion, but induction has to be assumed as an axiom.)

  Inductive types are an extra feature which are present in all widely used type-theoretic theorem provers, like Coq, Lego or Agda.

- Another feature that is not present in PTS, is the notion of (strong) **sigma type**. A Σ-type is a "dependent product type" and therefore a generalization of product type in the same way that a Π-type is a generalization of the arrow type.

  $\Sigma x{:}A.\,B$ represents the type of pairs (a, b) with $a : A$ and $b : B[x := a]$.

  (If $x \notin FV(B)$ we just end up with $A \times B$.)

  Note that products can be defined inside PTS with polymorphism, but Σ-type cannot.

Sigma types

$\Sigma x{:}A.\,B$ is the type of pairs $\langle a, b\rangle_{\Sigma x A.\,B}$ such that $a : A$ and $b : B[x := a]$.

Note that pairs are labeled with their types, so as to ensure uniqueness of types and decidability of type checking.

Besides the paring construction to create elements of a Σ-type, on also has projections to take a pair apart.

Extending PTS with Σ-types

- The set of pseudo-terms is extended as follows:

$$\mathcal{T} \ ::= \ \ldots \mid \Sigma\mathcal{V}{:}\mathcal{T}.\mathcal{T} \mid \langle \mathcal{T}, \mathcal{T}\rangle_{\mathcal{T}} \mid \mathsf{fst}\,\mathcal{T} \mid \mathsf{snd}\,\mathcal{T}$$

- **π-reduction** is defined by the contraction rules

$$\begin{array}{lll} \mathsf{fst}\,\langle M, N\rangle_{\Sigma x A.\,B} & \rightarrow_\pi & M \\ \mathsf{snd}\,\langle M, N\rangle_{\Sigma x A.\,B} & \rightarrow_\pi & N \end{array}$$

(cont.)

Sigma types

Extending PTS with Σ-types (cont.)

- The notion of specification is extended with a set $\mathcal{U} \subseteq S \times S \times S$ of rules for Σ-types.

  As usual, we use $(s1,s2)$ as an abbreviation for $(s1,s2,s2)$.

- The typing system is extended with the rules in the next slide. Moreover, the conversion rule is modified so as to include π-conversion.

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

(cont.)

Sigma types

Extending PTS with Σ-types (cont.)

$$(\text{sigma}) \qquad \frac{\Gamma \vdash A : s_1 \quad \Gamma, x{:}A \vdash B : s_2}{\Gamma \vdash (\Sigma x{:}A.\,B) : s_3} \qquad \text{if } (s_1, s_2, s_3) \in \mathcal{U}$$

$$(\text{pair}) \qquad \frac{\Gamma \vdash M : A \quad \Gamma \vdash N : B[x := M] \quad \Gamma \vdash (\Sigma x{:}A.\,B) : s}{\Gamma \vdash \langle M, N\rangle_{\Sigma x A.\,B} : (\Sigma x{:}A.\,B)}$$

$$(\text{proj1}) \qquad \frac{\Gamma \vdash M : (\Sigma x{:}A.\,B)}{\Gamma \vdash \mathsf{fst}\,M : A}$$

$$(\text{proj2}) \qquad \frac{\Gamma \vdash M : (\Sigma x{:}A.\,B)}{\Gamma \vdash \mathsf{snd}\,M : B[x := \mathsf{fst}\,M]}$$

A Σ-type as an existential quantification

Let us consider an extension of λPREDω with Σ-types.

Example: Assume we have the rule (Set, Prop, Prop) for Σ-types. One can have

$$N : \mathsf{Set}, \mathsf{Prime} : N \to \mathsf{Prop} \;\vdash\; (\Sigma n{:}N.\,\mathsf{Prime}\, n) : \mathsf{Prop}$$

This rule captures a form of existential quantification:

We can extract from a proof p of $\Sigma n{:}N.\ \mathsf{Prime}\ n$, read as "there exists a prime number n", both a witness (fst p) of type N and a proof (snd p) that (fst p) is prime.

A Σ-type as a "subset"

Assume we have the rule (Set, Prop, Type^p) for Σ-types.

This rule allows to form "subsets" of kinds. Combined with the rule (Set,Type^p,Type^p) this rule allows to introduce types of **algebraic structures**.

Example: Given a set A : Set, a monoid over A is a tuple consisting of

$\circ : A \to A \to A$, a binary operator

$\mathsf{e} : A$, the neutral element

such that the following types are inhabited

$$\Pi\, x, y, z : A.\, (x \circ y) \circ z =_L x \circ (y \circ z)$$
$$\Pi\, x : A.\, \mathsf{e} \circ x =_L x$$

A Σ-type as a "subset" (cont.)

The type of monoids over A, Monoid(A), can be defined by

$$\begin{aligned}\mathsf{Monoid}(A) \;:=\; & \Sigma \circ{:}A \to A \to A.\, \Sigma \mathsf{e} : A.\\ & (\Pi\, x, y, z{:}A.\, (x \circ y) \circ z =_L x \circ (y \circ z)) \wedge\\ & (\Pi\, x{:}A.\, \mathsf{e} \circ x =_L x)\end{aligned}$$

Conjunction and equality are define as described before.

If m :Monoid(A), we can extract the elements of the monoid structure by projections

$$\begin{aligned}\mathsf{fst}\, m &: A \to A \to A\\ \mathsf{fst}\,(\mathsf{snd}\, m) &: A\\ \mathsf{snd}\,(\mathsf{snd}\, m) &: \mathsf{MLaws}\, A\,(\mathsf{fst}\, m)\,(\mathsf{fst}\,(\mathsf{snd}\, m))\end{aligned}$$

assuming

$$\begin{aligned}\mathsf{MLaws} \;:=\; & \lambda A{:}\mathsf{Set}.\lambda \circ{:}A \to A \to A.\, \lambda \mathsf{e}{:}A.\\ & (\Pi\, x, y, z{:}A.\, (x \circ y) \circ z =_L x \circ (y \circ z)) \;\wedge\; (\Pi\, x{:}A.\, \mathsf{e} \circ x =_L x)\end{aligned}$$

Extended Calculus of Constructions

Extended Calculus of Constructions (ECC) is the underlying type theory of **Lego** proof assistant. It can be described by the follows

Extended Calculus of Constructions

Specification:

$$\begin{aligned}\mathcal{S} &= \mathsf{Prop},\ \mathsf{Type}_i \quad ,\ i \in \mathrm{N}\\ \mathcal{A} &= (\mathsf{Prop} : \mathsf{Type}),\ (\mathsf{Type}_i : \mathsf{Type}_{i+1}) \quad ,\ i \in \mathrm{N}\\ \mathcal{R} &= (\mathsf{Prop}, \mathsf{Prop}),\ (\mathsf{Prop}, \mathsf{Type}_i),\ (\mathsf{Type}_i, \mathsf{Prop}),\ (\mathsf{Type}_i, \mathsf{Type}_j, \mathsf{Type}_{\mathsf{max}(i,j)}) \quad ,\ i, j \in \mathrm{N}\\ \mathcal{U} &= (\mathsf{Prop}, \mathsf{Prop}, \mathsf{Prop}),\ (\mathsf{Type}_i, \mathsf{Type}_j, \mathsf{Type}_{\mathsf{max}(i,j)}) \quad ,\ i, j \in \mathrm{N}\end{aligned}$$

Cumulativity: $\mathsf{Prop} \subseteq \mathsf{Type}_0 \subseteq \mathsf{Type}_1 \subseteq \ldots$

In the current version of the **Coq** proof assistant, based on the Calculus of Inductive Constructions (CIC), the notion of Σ-type is implemented as an inductive type.

Inductive Types

Induction is a basic notion in logic and set theory.

- When a set is defined inductively we understand it as being "built up from the bottom" by a set of basic constructors.
- Elements of such a set can be decomposed in "smaller elements" in a well-founded manner.
- This gives us principles of:
    - **"proof by induction"** and
    - **"function definition by recursion"**.

Inductive Types

We can define a new type $\boldsymbol{I}$ inductively by giving its **constructors** together with their types which must be of the form

$$\boxed{\tau_1 \to \ldots \to \tau_n \to I \quad , \text{with} \;\; n \geq 0}$$

- Constructors (which are the introduction rules of the type $\boldsymbol{I}$) give the canonical ways of constructing one element of the new type .
- $\boldsymbol{I}$ defined is the smallest set (of objects) closed under its introduction rules.
- The inhabitants of type $\boldsymbol{I}$ are the objects that can be obtained by a finite number of applications of the type constructors.

NOTE: Type $\boldsymbol{I}$ can occur in any of the "domains" of its constructors. However, the occurrences of $\boldsymbol{I}$ in τ_i must be in **positive positions** in order to assure the well-foundedness of the datatype.

OK

$$\boldsymbol{I} \to B \to \boldsymbol{I}$$
$$A \to (B \to \boldsymbol{I}) \to \boldsymbol{I}$$
$$((\boldsymbol{I} \to A) \to B) \to A \to \boldsymbol{I}$$

Wrong !

$$(\boldsymbol{I} \to A) \to \boldsymbol{I}$$
$$((A \to \boldsymbol{I}) \to B) \to A \to \boldsymbol{I}$$

Examples

- The inductive type $\mathbb{N} : \mathsf{Set}$ of **natural numbers** has two constructors

$$0 : \mathbb{N}$$
$$\mathsf{S} : \mathbb{N} \to \mathbb{N}$$

- A well-known example of a higher-order datatype is the type $\mathbb{O} : \mathsf{Set}$ of ordinal notations which has three constructors

$$\begin{array}{lcl} \mathsf{Zero} & : & \mathbb{O} \\ \mathsf{Succ} & : & \mathbb{O} \to \mathbb{O} \\ \mathsf{Lim} & : & (\mathbb{N} \to \mathbb{O}) \to \mathbb{O} \end{array}$$

To program and reason about an inductive type we must have means to analyze its inhabitants.

The elimination rules for the inductive types express ways to use the objects of the inductive type in order to define objects of other types, and are associated to new computational rules.

Case analysis

The first elimination rule for inductive types one can consider is **case analyses**.

For instance, $n : \mathbb{N}$ means that n was introduced using either 0 or S, so we may define an object $\mathsf{case}\ n\ \mathsf{of}\ \{0 \Rightarrow b_1 \mid \mathsf{S} \Rightarrow b_2\}$ in another type σ depending on which constructor was used to introduce n .

A typing rule for this construction is

$$\frac{\Gamma \vdash n : \mathbb{N} \quad \Gamma \vdash b_1 : \sigma \quad \Gamma \vdash b_2 : \mathbb{N} \to \sigma}{\Gamma \vdash \mathsf{case}\ n\ \mathsf{of}\ \{0 \Rightarrow b_1 \mid \mathsf{S} \Rightarrow b_2\} : \sigma}$$

and the associated computing rules are

$$\begin{array}{lcl} \mathsf{case}\ 0\ \mathsf{of}\ \{0 \Rightarrow b_1 \mid \mathsf{S} \Rightarrow b_2\} & \to & b_1 \\ \mathsf{case}\ (\mathsf{S}\,x)\ \mathsf{of}\ \{0 \Rightarrow b_1 \mid \mathsf{S} \Rightarrow b_2\} & \to & b_2\,x \end{array}$$

The case analysis rule is very useful but it does not give a mechanism to define recursive functions.

Recursors

When an inductive type is defined in a type theory the theory should automatically generate a scheme for proof-by-induction and a scheme for primitive recursion.

- The inductive type comes equipped with a **recursor** that can be used to define functions and prove properties on that type.
- The recursor is a constant $\mathbf{R}_I$ that represents the structural induction principle for the elements of the inductive type $\boldsymbol{I}$, and the computation rule associated to it defines a safe recursive scheme for programming.

For example, $\mathbf{R}_\mathbb{N}$, the recursor for $\mathbb{N}$, has the following typing rule:

$$\frac{\Gamma \vdash P : \mathbb{N} \to \mathsf{Type} \quad \Gamma \vdash a : P\,0 \quad \Gamma \vdash a' : \Pi x{:}\mathbb{N}.\, P\,x \to P\,(\mathsf{S}\,x)}{\Gamma \vdash \mathbf{R}_\mathbb{N}\,P\,a\,a' : \Pi n{:}\mathbb{N}.\, P\,n}$$

and its reduction rules are

$$\begin{array}{lcl} \mathbf{R}_\mathbb{N}\,P\,a\,a'\,0 & \to & a \\ \mathbf{R}_\mathbb{N}\,P\,a\,a'\,(\mathsf{S}\,x) & \to & a'\,x\,(\mathbf{R}_\mathbb{N}\,P\,a\,a'\,x) \end{array}$$

Proof-by-induction scheme

The proof-by-induction scheme can be recovered from $\mathbf{R}_\mathbb{N}$ by setting P to be of type $\mathbb{N} \to \mathsf{Prop}$.

Let $\mathsf{ind}_\mathbb{N} := \lambda P{:}\mathbb{N} \to \mathsf{Prop}.\, \mathbf{R}_\mathbb{N}\,P$. We obtain the following rule

$$\frac{\Gamma \vdash P : \mathbb{N} \to \mathsf{Prop} \quad \Gamma \vdash a : P\,0 \quad \Gamma \vdash a' : \Pi x{:}\mathbb{N}.\, P\,x \to P\,(\mathsf{S}\,x)}{\Gamma \vdash \mathsf{ind}_\mathbb{N}\,P\,a\,a' : \Pi n{:}\mathbb{N}.\, P\,n}$$

This is the well known structural induction principle over natural numbers. It allows to prove some universal property of natural numbers $(\forall n{:}\mathbb{N}.\, Pn)$ by induction on n.

Primitive recursion scheme

The primitive recursion scheme (allowing dependent types) can be recovered from $\mathbf{R}_\mathbb{N}$ by setting P to be of type $\mathbb{N} \to \mathsf{Set}$.

Let $\mathsf{rec}_\mathbb{N} := \lambda P{:}\mathbb{N} \to \mathsf{Set}.\, \mathbf{R}_\mathbb{N}\,P$. We obtain the following rule

$$\frac{\Gamma \vdash T : \mathbb{N} \to \mathsf{Set} \quad \Gamma \vdash a : T\,0 \quad \Gamma \vdash a' : \Pi x{:}\mathbb{N}.\, T\,x \to T\,(\mathsf{S}\,x)}{\Gamma \vdash \mathsf{rec}_\mathbb{N}\,T\,a\,a' : \Pi n{:}\mathbb{N}.\, T\,n}$$

We can define functions using the recursors.

Example: A function that doubles a natural number can be defined as follows

$$\mathsf{double} := \mathsf{rec}_\mathbb{N}\,(\lambda n{:}\mathbb{N}.\,\mathbb{N})\,0\,(\lambda x{:}\mathbb{N}.\,\lambda y{:}\mathbb{N}.\,\mathsf{S}\,(\mathsf{S}\,y))$$

This gives us a safe way to express recursion without introducing non-normalizable objects. However, codifying recursive functions in terms of elimination constants can be rather difficult, and is quite far from the way we are used to program.

General recursion

Functional programming languages feature general recursion, allowing recursive functions to be defined by means of pattern-matching and a general fixpoint operator to encode recursive calls.

The typing rule for $\mathbb{N}$ fixpoint expressions is

$$\frac{\Gamma \vdash \mathbb{N} \to \theta : s \quad \Gamma, f : \mathbb{N} \to \theta \vdash e : \mathbb{N} \to \theta}{\Gamma \vdash (\mathsf{fix}\ f = e) : \mathbb{N} \to \theta}$$

and the associated computation rules are

$$\begin{array}{lcl} (\mathsf{fix}\ f = e)\,0 & \to & e[f := (\mathsf{fix}\ f = e)]\,0 \\ (\mathsf{fix}\ f = e)\,(\mathsf{S}\,x) & \to & e[f := (\mathsf{fix}\ f = e)]\,(\mathsf{S}\,x) \end{array}$$

Using this, the function that doubles a natural number can be defined by

$$(\mathsf{fix\ double} = \lambda\,n.\,\mathsf{case}\ n\ \mathsf{of}\ \{0 \Rightarrow 0 \mid \mathsf{S} \Rightarrow (\lambda x.\,\mathsf{S}\,(\mathsf{S}\,(\mathsf{double}\,x)\})$$

But, this approach opens the door to the introduction of non-normalizable objects.