From Types to Type Theory to Proofs of Programs

Jean-Pierre Jouannaud

Université Paris Saclay and Ecole Polytechnique, France

JAIST, March 7, 2016.

Outline

Outline

Outline

Outline

Outline

Birth of types in programming

Early programming languages were machine oriented. There were no types except those wired in the hardware.

The notions of type and type declaration in programming appeared with high level languages:

- Fortran (John Backus): integers, reals, arrays
- LISP (John Mac Carthy): the very first functional programming language had a single (implicit) type: lists
- ALGOL 68 (IFIP working group 2.1): all data are typed

Types belonged to pragmatics of programming languages.
Types had no intended logical meaning.
Programming languages aimed at structuring calculations, while types aimed at structuring data.

Birth of strongly typed languages [Robin Milner]

- Syntactic errors are caught by the syntax analyzer
- Design errors cannot be caught
- Common belief: machine overflows cannot be caught
- Other runtime errors should be caught by a type system: ML was the very first language in which the user could declare his/her own types in order to build a type system.

A simple ML-like program

```
Inductive nat : set :=
| O : nat
| S : nat → nat
end.
Fixpoint + (p : nat) (q : nat) : nat :=
   match p with
   | 0 ⇒ q
   | S n ⇒ S (+ n q)
end.
Inductive list : set :=
| nil : list
| cons : nat → list → list
end.
Fixpoint add (l : list): nat :=
match l with
   | nil ⇒ 0
   | cons head tail ⇒ + head (add tail)
end.
```

Types in ML

- nat, int, and list are types
- types have themselves a type, named set
- set has no type
- ML has function types: nat $\rightarrow$ nat,
  nat $\rightarrow$ nat $\rightarrow$ nat, nat $\rightarrow$ (nat $\rightarrow$ nat), (nat $\rightarrow$ nat) $\rightarrow$ nat, ...
- every expression is typed:
  S : nat $\rightarrow$ nat
  + : nat $\rightarrow$ nat $\rightarrow$ nat *or* nat $\rightarrow$ (nat $\rightarrow$ nat)
  p,q : nat
  S n : nat
  n : nat
  + n : nat $\rightarrow$ nat
  + n q : nat
  S(+ n q) : nat
- Theorem: a well-typed ML program cannot go wrong:
  either it terminates with a result of the appropriate type,
  or it does not terminate.

A more advanced ML-like program

```
Variable a : set.
Inductive list: a → set :=
| nil : list a
| cons : a → (list a) → (list a)
end.

Fixpoint add (l : list a) (0 : a) (+ : a → a → a) : a :=
match l with
   | nil ⇒ 0
   | cons head tail ⇒ + head (add tail)
end.

add : (list a) → a → (a → a → a) → a
```

The program add is polymorphic: it can apply to lists of nat, lists of int, lists of reals, list of Booleans, ...

Typing tells us more than the absence of runtime errors: if the first input of add has type (list a), its output has type a.

An even more advanced ML-like program

```
Inductive alph : set := a, b, c, d , ..., z .
Inductive word : nat → set :=
| epsilon : word 0
| char : alph → word 1
| append : forall (n p : nat), word n → word p → word (n + p) .
Fixpoint reverse (n : nat) (w : word n) : word n :=
match w with
   | epsilon ⇒ epsilon
   | char c ⇒ char c
   | append n1 n2 w1 w2 ⇒ append n2 n1 (reverse n2 w2)
                                       (reverse n1 w1)
end .
```

Typing tells us a more sophisticated property of reverse:
reversing a word does not change its length.

Heaven for programmers

Could typing tell us arbitrary properties of functional programs?

The answer is YES
It requires a logical system
called Type Theory

Outline

Propositions as types, Proofs as programs [Curry-Howard-De Bruiyn-Martin Löf-Girard]

- Every expression should have a type, including types, types of types, ...
- Any logical formula is a type and vice-versa
- Showing that a program P has type Q is the same as showing that program P has property Q:
  P becomes an element, (an inhabitant, a witness, a proof) of Q.
- In other words, the set of elements of a type is identified with the set of its proofs [Goedel]
- There is a certain type, representing Falsity, which cannot be inhabited.

Natural deduction rules for minimal logic [Goedel]

Propositions: $\alpha, \beta := p \in \{p_i\}_i \mid \alpha \to \beta$

$\Gamma \vdash P$ means P holds under assumptions in Γ

$$\frac{P \in \Gamma}{\Gamma \vdash P}\ [\textsc{Axiom}]$$

$$\frac{\Gamma, P \vdash Q}{\Gamma \vdash P \to Q}\ [\textsc{Intro}]$$

$$\frac{\Gamma \vdash P \to Q \qquad \Gamma \vdash P}{\Gamma \vdash Q}\ [\textsc{Elim}]$$

Curry: same as Church's typing rules

Type-checking rules for simply typed lambda-calculus

Types: $\alpha, \beta := p \in \{p_i\}_i \mid \alpha \to \beta$

$$\frac{x : P \in \Gamma}{\Gamma \vdash x : P}\ [\textsc{Var}]$$

$$\frac{\Gamma, x : P \vdash u : Q}{\Gamma \vdash \lambda[x : P].\, u : P \to Q}\ [\textsc{Abst}]$$

$$\frac{\Gamma \vdash v : P \to Q \qquad \Gamma \vdash w : P}{\Gamma \vdash (v\ w) : Q}\ [\textsc{App}]$$

Proof terms: $u, v, w := x : \alpha \mid \lambda x : \alpha.w \mid (u\ v)$

Howard: cut elimination is functional evaluation.

Polymorphism, dependent types, universes

Each syntactic construct (such as dependent typing) comes along with its two kinds of typing rules, called introduction and elimination. Introduction tells you if a given (dependent type) construction is well-formed, and elimination allows you to get information from a well-formed (dependent type) construction:

$$\frac{\Gamma \vdash t : T \qquad \Gamma \vdash u : U}{\Gamma \vdash \langle t, u \rangle : T \wedge U} \text{ [INTRO-AND]}$$

$$\frac{\Gamma \vdash \langle t, u \rangle : T \wedge U}{\Gamma \vdash t : T} \text{ [ELIM-LAND]}$$

Home work: elimination rule for $\vee$

And there is one more rule ...

Equality of dependent types

Types list (+ 0 2), list (+ 1 1), list (+ 2 0) and list 2 are extensionaly equal in the sense that they all characterize lists of length 2, hence have the same inhabitants.

These types are identified in type systems with dependent types by the conversion rule:

$$\frac{\Gamma \vdash t : T \qquad \Gamma \vdash T' : s' \qquad T \overset{\beta}{\longleftrightarrow}_* T'}{\Gamma \vdash t : T'} \text{ [Conv]}$$

Types list (+ n 0) and list n are not identified when n is a variable, althout they are extensionaly equal too.

This is the main weakness of Martin-Löf Type Theory

Type checking Reverse in Coq

```
Inductive nat : Set :=
| O : nat
| S : nat → nat .
Fixpoint + (p : nat) (q : nat) : nat := match q with
   | 0 ⇒ p
   | S n ⇒ S (+ p n)
end.
Variable A : Set .
Inductive word : nat → Set :=
| epsilon : word 0
| char : A → word 1
| append : forall (n p : nat),
   word n → word p → word (n + p) .
Fixpoint reverse (n : nat) (w : word n) : word n :=
match w with
   | epsilon ⇒ epsilon
   | char c ⇒ char c
   | append n1 n2 w1 w2 ⇒ append n2 n1 (reverse n2 w2)
                                       (reverse n1 w1)
end .
```

FAILS !

Extensionality

In Coq, it is possible to equip the previous definition so that the type checker generates the verification condition
$list\ (+\ n1\ n2) = list\ (+\ n2\ n1)$
to be carried out (in Coq) by the user.

This has severe drawbacks:

- the type-checker is not a stand-alone program anymore,
- This pollutes the proofs of properties of such definitions.
- It is unatural

Alternative: make CONV stronger by declaring a decidable equality for nat such as Presburger arithmetic, or the universal fragment of Peano arithmetic.

Reverse in CoqMT

```
Inductive word : nat → Set :=
| epsilon : word 0
| char : alph → word 1
| append : forall (n p : nat), word n → word p → word (n + p) .
Fixpoint reverse (n : nat) (w : word n) : word n :=
match w with
    | epsilon ⇒ epsilon
    | char c ⇒ char c
    | append n1 n2 w1 w2 ⇒ append n2 n1 (reverse n2 w2)
                                        (reverse n1 w1)
end .
Lemma reverse-is-involutive :
forall n (w : word n), reverse (reverse w) = w .
Proof . intros w; induction w; auto. auto. rewrite with IHw1;
rewrite with IHw2 ; trivial. Qed .
```

Isomorphicality

CoqMT solves the problem of extensionaly equal simple types, but does not solve the problem of isomorphic simple types.

Inductive nat : Set :=
| 0 : nat
| S : nat → nat .
Inductive Hnat : Set :=
| H0 : Hnat
| HS : Hnat → Hnat .
Inductive posnat : nat → Set := ....
Home work: complete the definiton.

nat, Hnat and posnat are isomorphic structures.
In HTT, they are identified by homotopic conversion:

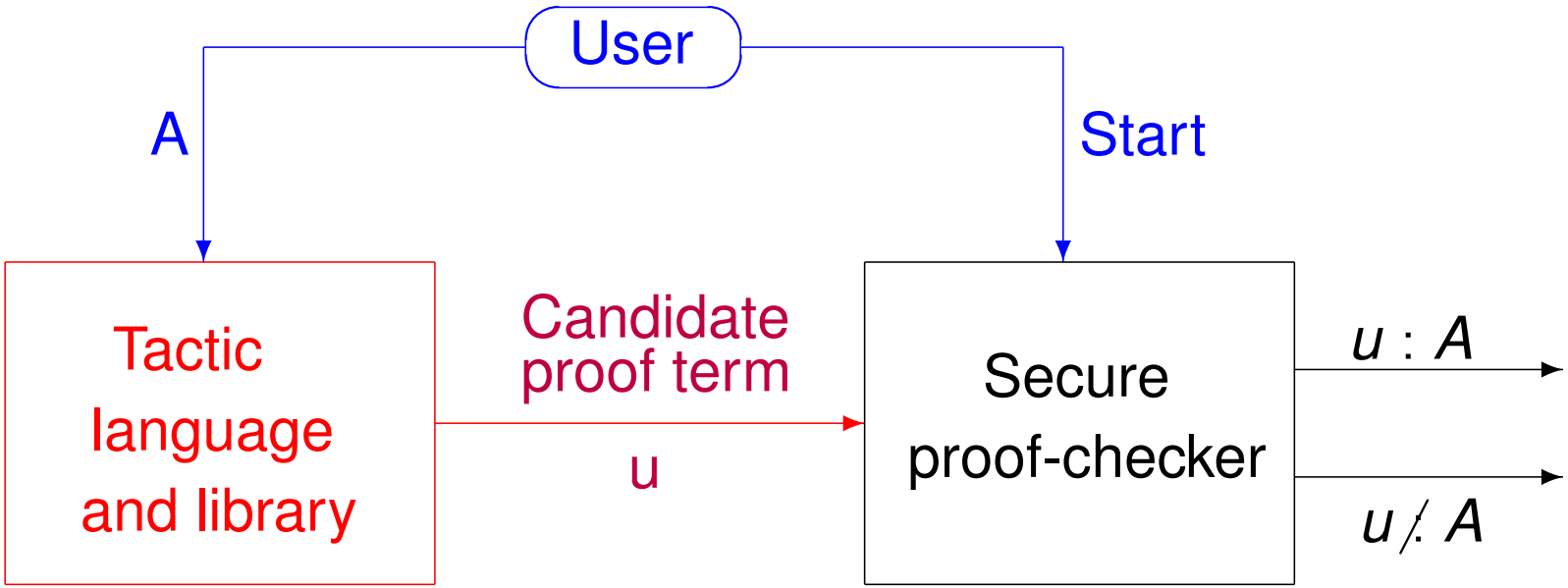
$$\frac{\Gamma \vdash t : T \qquad \Gamma \vdash T' : s' \qquad T \equiv T'}{\Gamma \vdash t : T'} \text{ [Conv]}$$

From earth to Heaven: a logic of computable functions

- Simply typed lambda calculus [Church]
- Goedel's system T [Goedel]
- Curry-Howard isomorphism [Curry, Howard]
- De Bruiyn's Automath [De Bruiyn]
- Martin Löf theory of types [Martin Löf]
- System F [Girard]
- Calculus of Constructions [Coquand, Huet]
- Calculus of Inductive Constructions [Coquand, Paulin-Mohring]
- Calculus of Extended Constructions [Luo]
- Calculus of Inductive Constructions [Werner]
- Calculus of Inductive Constructions modulo Theory [Barras, Jouannaud, Strub, Wang]
- Homotopy Type Theory [Hofman&Streicher, Voevodsky]

Outline

Architecture of Proof assistants [Milner, LCF]



Using PAs

Credo 1: PAs meet Software Verification needs
Compcert-C verified in Coq [Leroy]

Credo 2: Mathematics is the benchmark for PAs
Feit-Thomson proved in Coq [Gonthier]

Credo 3: PAs must be certified to be trusted
Coq [Barras] and CoqMT [Wang]

Coq has passed all these tests [ACM award]

but we need one more:

Credo 4: Modelization should avoid encodings

Prove your software S satisfies property P

First: write a model of S in your PA

- Method 1: prove the model satisfies $\forall x \exists y . P(x, y)$
- Method 2: given input i, compute o and prove $P(i, o)$

To enable method 2:

- the model of S should be executable
- the model of S should return a certificate describing how o is derived from i and this description should then be transformed into a proof term

Method 2 has become very popular:

- Done for Presburger arithmetic, SAT, SMT, termination analysis, arrays, etc.
- Annual competitions of certified provers
- Normalized certificates for SAT, termination

because it allows compare & reuse proofs via their certificates.

Prove your C software S satisfies property P

- step 1: instrument the program with appropriate logical formulas, including those that your program should verify
- step 2: automate model production by computing a translation of your program in the Coq language, and generate verification conditions implying that the transformation preserves the semantics of C.
- step 3: send the verification conditions to provers (Vampire, etc.), SMT solvers (Z3, CVC-4, Alt-Ergo, etc.), and PAs (Coq, HOL, etc.). (preferably returning Coq proof terms)
- step 4: type-check the obtained Coq program

Done in FRAMA-C at CEA and in their associated start-up company on a semi-industrial scale
[Filliatre, Monate, F. Kirchner]
Application are in aeronotics.

More on natural encodings

- Type theory is not enough: HOL better than Coq !
- Algebra is needed as in OBJ, MAUDE, Cafe-OBJ, ELAN, ...
- Functional encodings are needed: algebra is not enough !
- Type theory + Algebra: $\lambda\Pi$PModulo and Dedukti
- Natural encodings provide portability of proofs

Outline

Conclusion I: the lessons

- Formal proofs is a widely accepted concept
- Modelling is most important, PA-dependent
- Proof development is demanding but rewarding
- A PA is instrumental to carry out proofs when
  - structurally complex or
  - computationally demanding
- Proof checking may be demanding as well
- Trained engineers can now use secure PAs

Conclusion II: the questions

- Will homotopy type theory be a success
- Will PAs based on HTT be a success
- Will unifying formalisms emerge that provide a "universal" notion of proof term: $\lambda\Pi$Modulo [Dowek]

- Can Curry-Howard be extended to other programming paradigms [Krivine]
- Will these extensions yield successful PAs
- Do we have the right tactic languages: SSReflect [Gonthier]

The Coq project started at INRIA in 1983 by Gérard Huet. Work in all these directions is too preliminary to be conclusive.