Logical frameworks

Define proof-checkers (proof processing systems...) not specific to one theory (the Calculus of constructions, Set theory...) but where you can define your own theory

- Define your theory
- Check proofs expressed in this theory
What for?

- Re-checking proofs developed in other systems
- Interoperability
- Sustainability of libraries
is a language to express statements and proofs (a proof format) implemented in several systems: Dkcheck, Lambdapi, Kocheck...

All proofs welcome (built with resolution-based systems, tableaux-based ones, interactive ones...)
No such thing as a Coq proof, a PVS proof...
But proofs in a theory $T$ or $U$...
A natural formalism: neutral deduction
Why not use Predicate logic instead?

In Dedukti

- Function symbols can **bind** variables (like in $\lambda$-Prolog, Isabelle, The Edinburgh logical framework)
- **Proofs** are terms (like in The Edinburgh logical framework)
- Deduction and **computation** are mixed (like in Deduction modulo theory)
- Both constructive and classical proofs can be expressed (like in Ecumenical logic)

Reaps the benefits of several previous logical frameworks: $\lambda$-Prolog, Isabelle, The Edinburgh logical framework, Pure type systems, Deduction modulo theory, Ecumenical logic
The two features of Dedukti

Dedukti is a typed λ-calculus with

- Dependent types
- Computation rules

No typing rules today, but illustration of these features with examples
What is a theory?

In Predicate logic: a language (sorts, function symbols, and predicate symbols), and a set of axioms

In Dedukti: a set of symbols (replaces sorts, function symbols, predicate symbols, and axioms), and a set of computation rules
Predicate logic as a theory in Dedukti

Predicate logic is a sophisticated framework with notions of sort, function symbol, predicate symbol, arity, variable, term, proposition, proof...

A typed λ-calculus is much more primitive

These notions must be constructed
Terms and propositions: a first attempt

\( I : \text{TYPE} \)

function symbols: \( I \to \ldots \to I \to I \)

\( \text{Prop} : \text{TYPE} \)

predicate symbols: \( I \to \ldots \to I \to \text{Prop} \)

\( \Rightarrow : \text{Prop} \to \text{Prop} \to \text{Prop} \)

\( \forall : (I \to \text{Prop}) \to \text{Prop} \)

- Symbol declarations only (no computation rules yet)
- Simply typed \( \lambda \)-calculus (no dependent types yet)
- Types are terms of type \( \text{TYPE} \)
- \( \forall \) binds (higher-order abstract syntax: \( \forall x A \) expressed as \( \forall \lambda x A \))
Works if we want one sort

But if we want several (like in geometry: points, lines, circles...)

$I_1 : \text{TYPE}$

$I_2 : \text{TYPE}$

$I_3 : \text{TYPE}$

Several (an infinite number of?) symbols and several (an infinite number of?) quantifiers

$\forall_1 : (I_1 \rightarrow \text{Prop}) \rightarrow \text{Prop}$

$\forall_2 : (I_2 \rightarrow \text{Prop}) \rightarrow \text{Prop}$

$\forall_3 : (I_3 \rightarrow \text{Prop}) \rightarrow \text{Prop}$
Making the universal quantifier generic

Something like
\[ \forall : \prod X : \text{TYPE}, ((X \to \text{Prop}) \to \text{Prop}) \]

But does not work for two reasons
- (a minor one) no dependent products on TYPE
- (a major one) many things in TYPE beyond \( I_1, I_2, \) and \( I_3 \) (e.g. Prop)
Making the universal quantifier generic

$I : \text{TYPE} \quad I_1 : \text{TYPE}, \ I_2 : \text{TYPE}, \ I_3 : \text{TYPE}$

$Set : \text{TYPE} \quad \iota_1 : \text{Set}, \ \iota_2 : \text{Set}, \ \iota_3 : \text{Set}$

$\iota : \text{Set} \quad El : \text{Set} \rightarrow \text{TYPE} \quad El \iota \rightarrow I \quad El \iota_1 \rightarrow I_1, \ El \iota_2 \rightarrow I_2, \ El \iota_3 \rightarrow I_3$

$El : \text{Set} \rightarrow \text{TYPE} \quad l_1 : \text{TYPE}, \ l_2 : \text{TYPE}, \ l_3 : \text{TYPE}$

$Prop : \text{TYPE} \quad \Rightarrow : \text{Prop} \rightarrow \text{Prop} \rightarrow \text{Prop} \quad \forall : \Pi x : \text{Set}, (El x \rightarrow \text{Prop}) \rightarrow \text{Prop}$

Uses dependent types and computation rules

Reminiscent of expression of Simple type theory in Predicate logic, universes à la Tarski...
Proofs

So far: terms and propositions. Now: proofs

Proofs are trees, they can be expressed in Dedukti

Curry-de Bruijn-Howard: \( P \Rightarrow P \) should be the type of its proofs
But not possible here \( P \Rightarrow P : Prop : TYPE \) is not itself a type

\[ Prf : Prop \rightarrow TYPE \]
mapping each proposition to the type of its proofs: \( Prf (P \Rightarrow P) : TYPE \)

Not all types are types of proofs (e.g. \( I, El \iota, Prop... \))
Proofs

Brouwer-Heyting-Kolmogorov: $\lambda x : (\text{Prf } P)$, $x$ should be a proof of $P \Rightarrow P$
But has type $(\text{Prf } P) \rightarrow (\text{Prf } P)$ and not $\text{Prf } (P \Rightarrow P)$
$\text{Prf } (P \Rightarrow P)$ and $(\text{Prf } P) \rightarrow (\text{Prf } P)$ must be identified

A computation rule

$$\text{Prf } (x \Rightarrow y) \rightarrow (\text{Prf } x) \rightarrow (\text{Prf } y)$$

In the same way

$$\text{Prf } (\forall x p) \rightarrow \Pi z : (\text{El } x), (\text{Prf } (p z))$$

The function $\text{Prf}$ is an injective morphism from propositions to types: it is the Curry-de Bruijn-Howard isomorphism
Connectives

So far: ⇒ and ∀ only

\( T, \bot, \neg, \land, \lor, \exists \) defined à la Russell

\( \land : Prop \rightarrow Prop \rightarrow Prop \)
\( \Prf(x \land y) \rightarrow \Pi z : Prop, ((\Prf x \rightarrow \Prf y \rightarrow \Prf z) \rightarrow \Prf z) \)
Classical connectives

So far: constructive deduction rules only
What if you want to express classical proofs (a logical framework ought to be neutral)

Ecumenical logic: constructive and classical disjunction are governed by different rules: they are different symbols (like inclusive and exclusive disjunction): \( \lor \) and \( \lor_c \)

\[ \Rightarrow_c, \land_c, \lor_c, \forall_c, \exists_c \] defined using negative translation as a definition

\[ \land_c : \text{Prop} \rightarrow \text{Prop} \rightarrow \text{Prop} \]

\[ \land_c \rightarrow \lambda x : \text{Prop}, \lambda y : \text{Prop}, ((\neg \neg x) \land (\neg \neg y)) \]

Also a symbol \( \text{Prf}_c \)
Translating proofs developed in other systems

Three sets of systems

- Those with an internal proof language (Automath-like): Coq, Matita, Agda...
  Zenon, Archsat, all those that produce TSTP proofs...
  **Translation** from one language to another

- Those with a small kernel with a small number of handles (LCF-like): HOL Light, Isabelle/HOL...
  all complex operations eventually lead up to actions on these handles
  **Instrumentation** of this small kernel

- All those that do not fit in the two previous sets: PVS...
  most theorem provers for
  Predicate logic, most SMT solvers...
  Instrument the full system?
  The **proof sketch** method
The proof sketch method

Instrument the full system but do not attempt to build a full proof directly

The system will replace $A \land \top$ with $A$, $A \lor (B \lor C)$ with $B \lor (A \lor C)$ a hundred times, do not try to keep up

Instead build a proof sketch: like a proof tree, but where each node is produced with a deduction rule a small proof from its children

$$
\begin{align*}
A \lor_c (B \lor_c C) & \quad D \lor_c \neg B \\
\hline
(A \lor_c C) \lor_c D
\end{align*}
$$

And transform the proof sketch into a proof tree in a second step (using less powerful but better instrumented systems)
The rise of re-checking

At the beginning of the project: interoperability
In which theory do we have a proof of the four color theorem?

But re-checking proofs in Predicate logic seems to be an equally important application domain

- proof search systems / SMT solvers are very complex systems where a bug is not unlikely
- these systems are called as bookends by more general systems (B, Coq...) and in the end we are not sure what has been proved and in which theory