Dedukti

Logical frameworks

Build 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

Dedukti

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 $\mathcal{T}$ or $\mathcal{U}$...
A natural formalism: neutral deduction

Why not use Predicate logic instead?

In Dedukti

- Function symbols can bind variables (like in λ-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: λ-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 : TYPE
function symbols: $I \to \ldots \to I \to I$

$Prop$: TYPE
predicate symbols: $I \to \ldots \to I \to Prop$
$\Rightarrow$: $Prop \to Prop \to Prop$
$\forall$: $(I \to Prop) \to Prop$

- Symbol declarations only (no computation rules yet)
- Simply typed λ-calculus (no dependent types yet)
- Types are terms of type 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 : TYPE

I_2 : TYPE

I_3 : TYPE

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

$\forall_1 : (I_1 \to Prop) \to Prop$

$\forall_2 : (I_2 \to Prop) \to Prop$

$\forall_3 : (I_3 \to Prop) \to Prop$

Making the universal quantifier generic

Something like
$\forall : \Pi X : \mathsf{TYPE}, ((X \to \mathit{Prop}) \to \mathit{Prop})$

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

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

$Set : \mathrm{TYPE}$

$\iota : Set$ $\quad$ $\iota_1 : Set, \iota_2 : Set, \iota_3 : Set$

$El : Set \to \mathrm{TYPE}$

$El\ \iota \longrightarrow I$ $\quad$ $El\ \iota_1 \longrightarrow I_1, El\ \iota_2 \longrightarrow I_2, El\ \iota_3 \longrightarrow I_3$

$Prop : \mathrm{TYPE}$

$\Rightarrow\ : Prop \to Prop \to Prop$

$\forall : \Pi x : Set, (El\ x \to Prop) \to Prop$

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 $P \Rightarrow P : \mathit{Prop} : \mathrm{TYPE}$ is not itself a type

$\mathit{Prf} : \mathit{Prop} \rightarrow \mathrm{TYPE}$
mapping each proposition to the type of its proofs: $\mathit{Prf}\,(P \Rightarrow P) : \mathrm{TYPE}$

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

Proofs

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

A computation rule

$$\mathit{Prf}\ (x \Rightarrow y) \longrightarrow (\mathit{Prf}\ x) \to (\mathit{Prf}\ y)$$

In the same way

$$\mathit{Prf}\ (\forall\ x\ p) \longrightarrow \Pi z : (\mathit{El}\ x), (\mathit{Prf}\ (p\ z))$$

The function *Prf* is an injective morphism from propositions to types: it is the Curry-de Bruijn-Howard isomorphism

Connectives

So far: $\Rightarrow$ and $\forall$ only

$\top$, $\bot$, $\neg$, $\wedge$, $\vee$, $\exists$ defined *à la* Russell

$\wedge : \mathit{Prop} \to \mathit{Prop} \to \mathit{Prop}$
$\mathit{Prf}\,(x \ \wedge \ y) \longrightarrow \Pi z : \mathit{Prop}, ((\mathit{Prf}\, x \to \mathit{Prf}\, y \to \mathit{Prf}\, z) \to \mathit{Prf}\, z)$

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): $\vee$ and $\vee_c$

$\Rightarrow_c, \wedge_c, \vee_c, \forall_c, \exists_c$ defined using negative translation as a definition
$\wedge_c : Prop \rightarrow Prop \rightarrow Prop$
$\wedge_c \longrightarrow \lambda x : Prop, \lambda y : Prop, ((\neg\neg x) \wedge (\neg\neg y))$

Also a symbol 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 no 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 $A \wedge \top$ with A, $A \vee (B \vee C)$ with $B \vee (A \vee C)$ 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

$$\frac{A \vee_c (B \vee_c C) \quad D \vee_c \neg B}{(A \vee_c C) \vee_c D}$$

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