ENSIIE/Samovar

Exprimer ses théories en Dedukti, le vérificateur de preuves universel

JFLA 2017

Guillaume Burel

vendredi 6 et samedi 7 janvier 2017

Dedukti

Vérificateur de preuve pour le $\lambda\Pi$-calcul modulo théorie

`http://dedukti.gforge.inria.fr/`

`http://dedukti.gforge.inria.fr/jfla2017/`

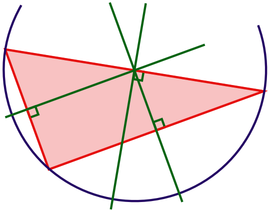
Dedukti : a Logical Framework based on the $\lambda\Pi$-Calculus Modulo Theory

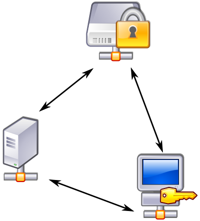
Ali Assaf, Guillaume Burel, Raphaël Cauderlier, David Delahaye, Gilles Dowek, Catherine Dubois, Frédéric Gilbert, Pierre Halmagrand, Olivier Hermant et Ronan Saillard.

`http://www.lsv.ens-cachan.fr/~dowek/Publi/expressing.pdf`

Sommaire

- Introduction
- Le $\lambda\Pi$-calcul modulo théorie
- Logiques des prédicats
- Langages de programmation
- Logique d'ordre supérieur
- Calcul des constructions inductives





Les démonstrations sont recherchées dans des **théories** :

- géométrie, arithmétique
- structures de données (listes chaînées, tableaux, . . .)
- propriétés du chiffrement
- . . .

Un cadre unique

Un seul cadre logique pour toutes les théories

- partage des connecteurs ($\wedge$, $\forall$, ...)
- partage des notions de preuve, de modèle
  - théorème généraux comme complétude prouvés une fois pour toute

- Notions d'inclusion entre théories (exemple ZF $\subset$ ZFC)

- interopérabilité $\mathcal{T}$ dans $\mathcal{L}$ $\qquad$ $\mathcal{T}'$ dans $\mathcal{L}'$ $\qquad$ $A \in \mathcal{L} \cap \mathcal{L}'$

$$\dfrac{\mathcal{T} \vdash A \Rightarrow B \qquad \mathcal{T}' \vdash A}{\mathcal{T} \cup \mathcal{T}' \vdash B}$$

Quel cadre utiliser ?

Logique du premier ordre ?

- Pour définir une théorie :
  - symboles de fonction et de prédicat
  - axiomes

Mais pas unique en pratique :

- théorie simple des types (HOL)
- calcul des constructions inductives

Inconvénient

Pas de possibilité de passer d'un prouveur à un autre

- CompCert en Isabelle/HOL ?
- seL4 en Coq ?

Combinaison avec des outils de preuve automatique ?

- éviter la reconstruction de preuve

Pourquoi ?

Manque en logique du premier ordre :

1. Possibilité de définir des lieurs
   - exemple $x \mapsto t$
2. Pas de correspondance preuve/programme — formule/type
3. Pas de prise en compte du calcul
   - simulé par des étapes de déduction
4. Pas de notion universelle de coupure
   - doit être définie pour chaque théorie
5. Cadres différents pour logiques classique et intuitionniste

Résoudre problèmes 1 et 2

Logical Framework LF a.k.a. λP a.k.a. $\lambda\Pi$ [Harper Honsell Plotkin 1993]

Extension

- du λ-calcul simplement typé avec des types dépendants
- de la logique des prédicats (minimale) avec des termes de preuve

Résoudre problèmes 3 et 4

Déduction modulo théorie [Dowek Hardin Kirchner 2003]
Règles d'inférence appliquées modulo une congruence $\equiv$

Système de réécriture sur les termes et les propositions

Exemple : $X \in \mathcal{P}(Y) \to \forall Z.\ Z \in X \Rightarrow Z \in Y$

$$\vdash\forall\ \dfrac{\vdash\Rightarrow\ \dfrac{\overset{\frown}{\vdash}\ \dfrac{}{Z \in a \vdash Z \in a}}{\vdash Z \in a \Rightarrow Z \in a}}{\vdash a \in \mathcal{P}(a)}\ a \in \mathcal{P}(a) \equiv \forall Z.\ Z \in a \Rightarrow Z \in a$$

Résoudre 1, 2, 3 et 4

Combiner LF avec la déduction modulo théorie :

- $\lambda\Pi$-calcul modulo théorie

Extension de la règle de conversion

- sans modulo : uniquement β
- avec : β + système de réécriture

Problème 5

Un connecteur $\triangleright$ pour passer d'atome à formule

Deux versions des connecteurs : une constructive, une classique

Dedukti

$\lambda\Pi$-calcul modulo théorie pas seulement expressif en théorie

Implémentation efficace :

- Dedukti

vérificateur de preuve du $\lambda\Pi$-calcul modulo théorie

M. Boespflug 2008 $\rightsquigarrow$ ... $\rightsquigarrow$ Ronan Saillard 2015

Démo

Outils autour de Dedukti

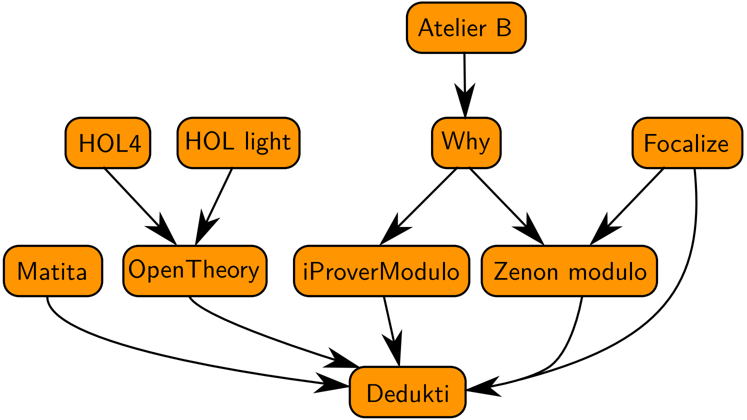
Traducteurs de preuves vers Dedukti

- HOL
- CIC

Backend vers Dedukti

- prouveurs automatiques iProverModulo, Zenon modulo
- focalize

Panorama actuel



En cours : Coq, PVS, VeriT, ...

Plongement profond vs. plongement superficiel

Superficiel : réutilisation des fonctionnalités du langage cible :

- contextes
- connecteurs et lieur
- calcul
- ...

Intérêt des plongements superficiels

- traductions plus légères
- plus efficace
- rend l'interopérabilité plus simple

Outline

- Introduction
- Le $\lambda\Pi$-calcul modulo théorie
- Logiques des prédicats
- Langages de programmation
- Logique d'ordre supérieur
- Calcul des constructions inductives

Le $\lambda\Pi$-calcul

$$t, u ::= Type \mid Kind \mid x \mid \lambda x : t.\ u \mid t\ u \mid \Pi x : t.\ u$$

niveaux :	$Kind$				
	$Type$		$\Pi x : nat.\ Type$		types des (familles de) types
	nat		$\Pi x : nat.\ vect\ x$		(familles) de types
	0	S	nil	$cons$	objets

Types dépendants

```
x : nat -> nat ✓
x : nat -> vect x ✓
x : nat -> Type ✓
A : Type -> list A X
A : Type -> Type X
```

$\lambda\Pi$-calcul modulo théorie

En plus des déclarations de variables
On va rajouter des règles de réécriture dans les contextes

$$l \longrightarrow^{\Delta} r$$

Δ : contexte local à la règle, ne contient que des déclarations de variables objets

$$\frac{\Gamma \vdash A : \mathit{Type} \quad \Gamma \vdash B : \mathit{Type} \quad \Gamma \vdash t : A}{\Gamma \vdash t : B} \; A \equiv_{\beta\Gamma} B$$

On peut réécrire avec les règles dans Γ

Bonnes hypothèses

Quelles hypothèses pour :

- réduction du sujet
- décidabilité de la vérification de type ?

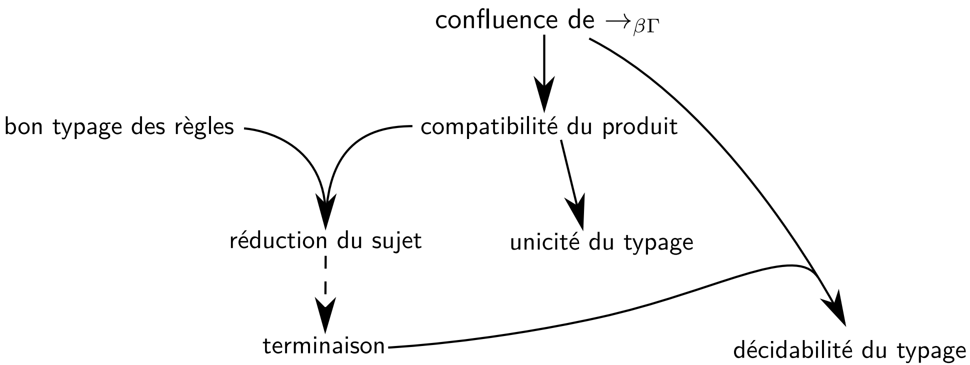
Definition (Bon typage d'une règle)

$l \longrightarrow^{\Delta} r$ est bien typé si :
Pour toute substitution σ des variables de Δ,
si $\Gamma \vdash \sigma l : T$ alors $\Gamma \vdash \sigma r : T$.

Definition (Compatibilité avec le produit)

si $\Pi x : A_1\ B_1 \equiv_{\beta\Gamma} \Pi x : A_2\ B_2$ then $A_1 \equiv_{\beta\Gamma} A_2$ and $B_1 \equiv_{\beta\Gamma} B_2$

Implications



Bon typage des règles

$l \longrightarrow^{\Delta} r$ bien typé si $\Gamma, \Delta \vdash l : T$ et $\Gamma, \Delta \vdash r : T$

mais ne marche pas pour $tail\ n\ (cons\ m\ a\ v) \longrightarrow^{n,m,a,v} v$ alors que bien typée

Il est suffisant :

- Pour la substitution la plus générale τ permettant de typer τl dans Γ,
  Si $\Gamma, \Delta \vdash \tau l : T$ alors $\Gamma, \Delta \vdash \tau r : T$

Pour calculer τ, besoin de savoir quels symboles sont injectifs

- `def`

Outline

- Introduction
- Le $\lambda\Pi$-calcul modulo théorie
- **Logiques des prédicats**
- Langages de programmation
- Logique d'ordre supérieur
- Calcul des constructions inductives

Logique des prédicats minimale

$$
\begin{array}{rcl}
t & ::= & x \mid f(t, \ldots, t) \\
A & ::= & P(t_1, \ldots, t_n) \mid A \Rightarrow A \mid \forall_{s_1} x\ A \mid \ \cdots\ \mid \forall_{s_n} x\ A
\end{array}
$$

Logique des prédicats minimale

$$
\begin{array}{rcl}
t & ::= & x \mid f(t, \ldots, t) \\
A & ::= & P(t_1, \ldots, t_n) \mid A \Rightarrow A \mid \forall_{s_1} x\ A \mid \cdots \mid \forall_{s_n} x\ A
\end{array}
$$

```
s1 : Type.
s2 : Type.
f : s1 -> s2.
P : s2 -> s1 -> Type.
```

$$
\begin{array}{c}
\|P(t_1, \ldots, t_n)\| = (\texttt{P}\ |t_1|\ \ldots\ |t_n|) \\
\|A \Rightarrow B\| = \|A\| \texttt{->} \|B\| \\
\|\forall_s x\ A\| = \texttt{x : s ->} \|A\|
\end{array}
$$

Logique des prédicats constructive

$$\begin{array}{rcl} t & ::= & x \mid f(t, \ldots, t) \\ A & ::= & P(t_1, \ldots, t_n) \mid A \Rightarrow A \mid \forall_s x\ A \mid \top \mid \bot \mid \neg A \mid A \wedge A \mid A \vee A \mid \exists_s x\ A \end{array}$$

Logique des prédicats constructive

$$
\begin{array}{rcl}
t & ::= & x \mid f(t, \ldots, t) \\
A & ::= & P(t_1, \ldots, t_n) \mid A \Rightarrow A \mid \forall_s x\ A \mid \top \mid \bot \mid \neg A \mid A \wedge A \mid A \vee A \mid \exists_s x\ A
\end{array}
$$

Étendre le $\lambda\Pi$-calcul modulo théorie

- avec de nouveaux connecteurs
- avec des inductifs ?

Logique des prédicats constructive

$$
\begin{array}{rcl}
t & ::= & x \mid f(t, \dots, t) \\
A & ::= & P(t_1, \dots, t_n) \mid A \Rightarrow A \mid \forall_s x\ A \mid \top \mid \bot \mid \neg A \mid A \wedge A \mid A \vee A \mid \exists_s x\ A
\end{array}
$$

Étendre le $\lambda\Pi$-calcul modulo théorie

- avec de nouveaux connecteurs
- avec des inductifs ?

Non, on peut rester avec le même base

Idée :

- Plongement profond dans un type o
- Remontée superficielle avec opérateur eps
  en utilisant encodage imprédicatif des connecteurs

Encodage imprédicatif

```
o : Type.
bot : o.
imp : o -> o -> o.
or : o -> o -> o.

def eps : o -> Type.

[a,b] eps (imp a b) --> eps a -> eps b.
```

Encodage imprédicatif

```
o : Type.
bot : o.
imp : o -> o -> o.
or : o -> o -> o.

def eps : o -> Type.

[a,b] eps (imp a b) --> eps a -> eps b.
[] eps bot --> z : o -> eps z.
```

Encodage imprédicatif

```
o : Type.
bot : o.
imp : o -> o -> o.
or : o -> o -> o.

def eps : o -> Type.

[a,b] eps (imp a b) --> eps a -> eps b.
[] eps bot --> z : o -> eps z.
[a,b] eps (or a b) -->
  z : o -> (eps a -> eps z) -> (eps b -> eps z) -> eps z.
```

iProverModulo

iProver [Korovin 2008]

- prouveur automatique pour la logique des prédicats classique
- basé (entre autres) sur la résolution
- écrit en OCaml

patché pour intégrer la déduction modulo théorie [Burel 2011]

capable de sortir des preuves Dedukti pour la partie résolution [Burel 2013]
(qui est constructive !)

Logique des prédicats classique

Idée : introduire des connecteurs différents

$$
\begin{array}{rcl}
t & ::= & x \mid f(t, \ldots, t) \\
a & ::= & P(t, \ldots, t) \\
A & ::= & a \mid A \Rightarrow A \mid \forall x\ A \mid \top \mid \bot \mid \neg A \mid A \wedge A \mid A \vee A \mid \exists x\ A \mid \\
 & & a \mid A \Rightarrow_c A \mid \forall_c x\ A \mid \top_c \mid \bot_c \mid \neg_c A \mid A \wedge_c A \mid A \vee_c A \mid \exists_c x\ A
\end{array}
$$

Logique des prédicats classique

Idée : introduire des connecteurs différents

$$
\begin{array}{rcl}
t & ::= & x \mid f(t, \dots, t) \\
a & ::= & P(t, \dots, t) \\
A & ::= & \triangleright\, a \mid A \Rightarrow A \mid \forall x\ A \mid \top \mid \bot \mid \neg A \mid A \wedge A \mid A \vee A \mid \exists x\ A \mid \\
 & & \triangleright_c\, a \mid A \Rightarrow_c A \mid \forall_c x\ A \mid \top_c \mid \bot_c \mid \neg_c A \mid A \wedge_c A \mid A \vee_c A \mid \exists_c x\ A
\end{array}
$$

Logique des prédicats classique

Idée : introduire des connecteurs différents

$$
\begin{array}{rcl}
t & ::= & x \mid f(t, \ldots, t) \\
a & ::= & P(t, \ldots, t) \\
A & ::= & \triangleright a \mid A \Rightarrow A \mid \forall x\ A \mid \top \mid \bot \mid \neg A \mid A \wedge A \mid A \vee A \mid \exists x\ A \mid \\
 & & \triangleright_c a \mid A \Rightarrow_c A \mid \forall_c x\ A \mid \top_c \mid \bot_c \mid \neg_c A \mid A \wedge_c A \mid A \vee_c A \mid \exists_c x\ A
\end{array}
$$

Définir les connecteurs classiques en fonction des intuitionnistes en utilisant une $\neg\neg$-traduction.

$$\triangleright_c a := \neg(\neg \triangleright a)$$

$$a \vee_c b := \neg(\neg(a \vee b))$$

Zenon Modulo

Extension de Zenon pour traiter la déduction modulo théorie [Halmagrand 2013]

Capable de produire des preuves Dedukti

600MB de preuves gzippées (à partir d'obligation de preuves B)

Outline

- Introduction
- Le $\lambda\Pi$-calcul modulo théorie
- Logiques des prédicats
- Langages de programmation
- Logique d'ordre supérieur
- Calcul des constructions inductives

λ-calcul non typé

$$app\ (lam\ f)\ t \longrightarrow^{f,t} f\ t$$

λ-calcul simplement typé

Plongement profond des types simples

```
type : Type.
o : type.
b : type.
arrow : type -> type -> type.
```

Remontée superficielle avec `term`

```
def term : type -> Type.
[a,b] term (arrow a b) --> term a -> term b.
```

ML

Utilisation de destructeurs à la place du filtrage de motifs

$\texttt{destr}_{\texttt{C}}$ `R (C t) f d --> f t`
$\texttt{destr}_{\texttt{C}}$ `R (C' t') f d --> d` (for all other constructors `C'` for type τ')

Blocage de la récursion :
`@ : A : type -> B: type -> ((eps A -> eps B) -> eps A -> eps B)`
`[R, f, t] @` τ' `R f (C t) --> f (C t)`

ς-calcul et FoCaLiZe

ς-calcul : langage pour la programmation orientée objet

FoCaLiZe : environnement pour le développement de programmes certifiés

- ML comme langage d'implémentation
- logique des prédicats classique comme langage de spécification
- aspects orientés objet pour la modularité

Outline

- Introduction
- Le $\lambda\Pi$-calcul modulo théorie
- Logiques des prédicats
- Langages de programmation
- Logique d'ordre supérieur
- Calcul des constructions inductives

Théorie simple des types

Langage :

- λ-calcul simplement typé
- types de base `bool` et `ind`
- constantes `imp : bool -> bool -> bool` et $\texttt{forall}_A$ `: (`A` -> bool) -> bool`

Plongement profond des types, des termes de type bool

Deux fonctions pour remonter

Théorie simple des types

Langage :

- λ-calcul simplement typé
- types de base `bool` et `ind`
- constantes `imp : bool -> bool -> bool` et `forall`$_A$ `: (`A `-> bool) -> bool`

Plongement profond des types, des termes de type bool

Deux fonctions pour remonter

En fait, tout est défini à l'aide de $=$

Outline

- Introduction
- Le $\lambda\Pi$-calcul modulo théorie
- Logiques des prédicats
- Langages de programmation
- Logique d'ordre supérieur
- Calcul des constructions inductives

Calcul des constructions

Deux traductions des termes :

- profonde, vus comme des termes $|t|$
- superficielle, vus comme des types $||t||$

plusieurs fonctions de remontée en fonction de la sorte

ϵ_{Type} et ϵ_{Kind}

Exemple

```
Definition id_Type := fun x : Type => x.
```

est traduit par

`id_Type :` $\|\mathsf{Type} \to \mathsf{Type}\|$`.`

`[] id_Type -->` $|\lambda x : \mathsf{Type}.\ x|$`.`

Exemple

```
Definition id_Type := fun x : Type => x.
```

est traduit par

id_Type : $\|$Type $\to$ Type$\|$.
[] id_Type --> x : $\|$Type$\|$ => x.

- λs traduits comme des λs

Exemple

```
Definition id_Type := fun x : Type => x.
```

est traduit par

id_Type : $\|$Type $\to$ Type$\|$.

```
[] id_Type --> x : U_Type => x.
```

- λs traduits comme des λs
- à chaque sorte s correspond un univers U_s

Exemple

```
Definition id_Type := fun x : Type => x.
```

est traduit par

```
id_Type : e_Kind |Type → Type|.
[] id_Type --> x : U_Type => x.
```

- λs traduits comme des λs
- à chaque sorte s correspond un univers `U_s`
- `e_s` : fonction de remontée pour s

Exemple

```
Definition id_Type := fun x : Type => x.
```

est traduit par

id_Type : e_Kind $|\Pi\ \_\ :\ \mathsf{Type}.\ \mathsf{Type}|$.

```
[] id_Type --> x : U_Type => x.
```

- λs traduits comme des λs
- à chaque sorte s correspond un univers U_s
- e_s : fonction de remontée pour s

Exemple

```
Definition id_Type := fun x : Type => x.
```

est traduit par

```
id_Type : e_Kind (pi_KK |Type| (fun _ : ||Type|| => |Type|)).
[] id_Type --> x : U_Type => x.
```

- λs traduits comme des λs
- à chaque sorte s correspond un univers U_s
- e_s : fonction de remontée pour s
- pi_KK : plongement profond des produits dépendants

Exemple

```
Definition id_Type := fun x : Type => x.
```

est traduit par

```
id_Type : w : e_Kind |Type|-> e_Kind((fun _ : ||Type|| => |Type|) w).
[] id_Type --> x : U_Type => x.
```

- λs traduits comme des λs
- à chaque sorte s correspond un univers `U_s`
- `e_s` : fonction de remontée pour s
- `pi_KK` : plongement profond des produits dépendants
- règles de réécriture pour remonter
  `e_Kind (pi_KK x y) --> w : e_Kind x -> e_Kind (y w).`

Exemple

```
Definition id_Type := fun x : Type => x.
```

est traduit par

```
id_Type : w : e_Kind |Type| -> e_Kind |Type|.
[] id_Type --> x : U_Type => x.
```

- λs traduits comme des λs
- à chaque sorte s correspond un univers `U_s`
- `e_s` : fonction de remontée pour s
- `pi_KK` : plongement profond des produits dépendants
- règles de réécriture pour remonter
  `e_Kind (pi_KK x y) --> w : e_Kind x -> e_Kind (y w).`

Exemple

```
Definition id_Type := fun x : Type => x.
```

est traduit par

```
id_Type : w : e_Kind u_Type -> e_Kind u_Type.
[] id_Type --> x : U_Type => x.
```

- λs traduits comme des λs
- à chaque sorte s correspond un univers `U_s` et une constante `u_s`
- `e_s` : fonction de remontée pour s
- `pi_KK` : plongement profond des produits dépendants
- règles de réécriture pour remonter
  `e_Kind (pi_KK x y) --> w : e_Kind x -> e_Kind (y w).`

Exemple

```
Definition id_Type := fun x : Type => x.
```

est traduit par

```
id_Type : w : U_Type -> U_Type.
[] id_Type --> x : U_Type => x.
```

- λs traduits comme des λs
- à chaque sorte s correspond un univers `U_s` et une constante `u_s`
- `e_s` : fonction de remontée pour s
- `pi_KK` : plongement profond des produits dépendants
- règles de réécriture pour remonter

  ```
  e_Kind (pi_KK x y) --> w : e_Kind x -> e_Kind (y w).
  e_Kind u_Type --> U_Type.
  ```

Inductifs

Cf. ML

Utilisation d'éliminateur au lieu de destructeurs + récursion

```
elim_list : A : Type
  -> P : (List A -> Type)
  -> P (nil A)
  -> (x : A -> l : list A -> P l -> P (cons A x l))
  -> l : list A
  -> P l.
```

Hiérarchie d'univers

Univers indexés par un niveau

- $\mathit{Type}_i : \mathit{Type}_{i+1}$
- $\mathit{Type}_i \subset \mathit{Type}_{i+1}$

Conversion explicite avec $\uparrow_i$: $\mathit{Type}_i \to \mathit{Type}_{i+1}$

Plus d'unicité du typage

- $\uparrow_i \Pi_i x : A.\ B$ vs. $\Pi_{i+1} x : \uparrow_i A.\ \uparrow_i B$

```
[i, a, b] pi _ (lift i a) (x => lift {i} (b x)) -->
          lift i (pi i a (x => b x)).
```