MDV : Typage

Plan

1 Introduction

Plan

1. Introduction
2. λ-calcul simplement typé

Plan

1. Introduction
2. λ-calcul simplement typé
3. Types dépendants

Plan

1. Introduction
2. λ-calcul simplement typé
3. Types dépendants
4. Polymorphisme

Plan

1. Introduction
2. λ-calcul simplement typé
3. Types dépendants
4. Polymorphisme
5. Constructeurs de types

Plan

1. Introduction
2. λ-calcul simplement typé
3. Types dépendants
4. Polymorphisme
5. Constructeurs de types
6. Conclusion

Vérification automatisée

Formaliser les notions de base, définitions, axiomes et preuves.

Choix d'une logique : classique, intuitionniste, premier ordre, ordre supérieur.

Choix de la formalisation des objets mathématiques

- théorie des ensembles (ex. Zermelo-Fraenkel avec axiome du choix)
  difficulté pour formaliser la notion de calcul
- théorie(s) des types
  formalise dans le même cadre les notions de calcul et de preuve
  - les preuves sont des « citoyens de première classe »
  - construction de la preuve : tactiques
  - vérification de la preuve : vérification de type (*type checking*)

Name dropping

- Brouwer & Heyting : logique intuitionniste
- Russel : notion de type
- Gentzen : déduction naturelle
- Church & Curry : λ-calcul typé
- Howard : propositions = types (PAT)
- de Bruijn : types dépendants
- Scott : types inductifs
- Martin-Löf : PAT avec types inductifs
- Girard : ordre supérieur
- Coquand & Huet : tout ça (calcul des constructions inductives)

propositions = types (Curry-Howard)

Types : un énoncé A peut être interprété par $[A]$, « collection » des preuves de A, c'est-à-dire un type.

$$[A \Rightarrow B] = [A] \longrightarrow [B]$$

fonctions qui, étant donnée une preuve de A, donnent une preuve de B

$$[A \wedge B] = [A] \times [B]$$

produit cartésien

Preuves :

$$\Gamma \vdash p : A$$

sous les hypothèses Γ, p est une preuve de A (on laisse tomber les $[\,]$)

Vérification de types :

$$\mathrm{Type}_\Gamma(p) = A$$

décidabilité de $=$: relation de conversion définie à partir d'un ensemble de réductions.

Trois problèmes

$\Gamma \vdash p : A?$	TCP, vérification de type
$\Gamma \vdash p :?$	TSP, synthèse de type
$\Gamma \vdash ? : A$	TIP, « habitation » de type

- TIP est indécidable pour toute théorie intéressante.
- TCP et TSP peuvent être décidables, en fonction des règles de typage et de la quantité d'information de typage donnée dans p.
  Même si on ne peut *trouver* une preuve de A, on peut en *reconnaître* une.

Critère de de Bruijn : les objets de preuve peuvent être vérifiés par un algorithme (vérificateur de types) simple (petit, et vérifiable à la main).

Plan

1. Introduction
2. λ-calcul simplement typé
3. Types dépendants
4. Polymorphisme
5. Constructeurs de types
6. Conclusion

λ-calcul typé à la Church

On se donne un ensemble infini dénombrable de variables.
Types simples : On se donne un ensemble de types de base (ex. $\{nat, bool\}$).
L'ensemble des types simples est défini inductivement par :

- les types de base sont des types,
- si A et B sont des types alors $(A \to B)$ est un type.

Termes typés : l'ensemble des termes est défini inductivement par :

- les variables sont des termes,
- si t et u sont des termes, alors $(t\ u)$ est un terme,
- si t est un terme, x une variable et A un type, alors $\lambda x : A,\ t$ est un terme.

On utilisera l'associativité implicite à droite pour les types et à gauche pour les termes applicatifs.

Contexte : une variable typée est un couple (x, A) ; un contexte Γ est une liste de variables typées, où chaque variable apparaît au plus une fois. L'ensemble des triplets (Γ, t, A) est bien typés est défini inductivement par :

- si $(x, A) \in \Gamma$ alors (Γ, t, A) est bien typé,
- si $(\Gamma, t, A \to B)$ et (Γ, u, A) sont bien typés alors $(\Gamma, (t\ u), B)$ est bien typé,
- si $(\Gamma[x : A], t, B)$ est bien typé alors $(\Gamma, \lambda x : A,\ t, A \to B)$ est bien typé.

On notera $\Gamma \vdash t : A$.

Proposition

Soit Γ un contexte et t un terme, il existe au plus un type A tel que $\Gamma \vdash t : A$.

Preuve : Par induction sur la dérivation de $\Gamma \vdash t : A$ □

Proposition

Il existe un algorithme qui prend un contexte Γ et un terme t et qui décide si t est typable dans Γ, et qui, si t est typable, retourne le type de t dans Γ.

Réductions

α-équivalence : équivalence induite par le renommage des variables liées n'introduisant pas de captures. Ex : $\lambda x : nat,\ x$ et $\lambda y : nat,\ y$ sont équivalents, mais $\lambda x : nat,\ \lambda y : nat,\ (f\ x\ y)$ et $\lambda x : nat,\ \lambda x : nat,\ (f\ x\ x)$ ne le sont pas.

β-réduction : un β-radical (redex) est un terme de la forme $((\lambda x : A,\ t)\ u)$. La β-réduction entre termes t et u, notée $t \rhd_\beta u$ est la plus petite relation telle que

- $((\lambda x : A,\ t)\ u) \rhd_\beta t[x \leftarrow u]$,
- si $t \rhd_\beta u$ alors $(t\ v) \rhd_\beta (u\ v)$,
- si $t \rhd_\beta u$ alors $(v\ t) \rhd_\beta (v\ u)$,
- si $t \rhd_\beta u$ alors $\lambda x : A,\ t \rhd_\beta \lambda x : A,\ u$.

La relation $\rhd_\beta^*$ est la fermeture réflexive-transitive de $\rhd_\beta$, et la relation de β-équivalence $\equiv_\beta$ est la fermeture réflexive-transitive-symétrique de $\rhd_\beta$.

Exercice : quelles sont les réductions possibles du terme $(\lambda f : nat \to nat \to nat,\ (\lambda x : nat, (f\ x)\ 2)\ \lambda\, x : nat,\ \lambda\, y : nat \to nat,\ (y\ x))$?

Réductions (suite)

η-réduction : on définit de même la η-réduction à partir de $\lambda x : A,\ (t\ x) \triangleright_\eta t$ si x n'apparaît pas dans t.

Proposition (Préservation du type)

Si $\Gamma \vdash t : A$ et $t \triangleright_{\beta\eta} t'$ alors $\Gamma \vdash t' : A$.

Preuve : Par induction sur la dérivation du type A □

Confluence

Une relation de réduction est dite confluente si, chaque fois qu'un terme t se réduit sur deux termes u_1 et u_2, il existe un terme v tel que u_1 et u_2 se réduisent sur v.

$$\begin{array}{ccc} t & \longrightarrow^* & u_1 \\ \downarrow^* & & \downarrow^* \\ u_2 & \longrightarrow^* & v \end{array}$$

Confluence de la βη-réduction :

- la β-réduction est confluente sur tous les termes (bien typés ou non) du λ-calcul simplement typé,
- pour la η-réduction, il faut adapter la démonstration de la confluence du λ-calcul pur en considérant uniquement les termes bien typés.
  Ex : le terme $\lambda x : A,\ (\lambda y : B,\ y\ x)$

Confluence

Une relation de réduction est dite confluente si, chaque fois qu'un terme t se réduit sur deux termes u_1 et u_2, il existe un terme v tel que u_1 et u_2 se réduisent sur v.

$$\begin{array}{ccc} t & \longrightarrow^* & u_1 \\ \downarrow^* & & \downarrow^* \\ u_2 & \longrightarrow^* & v \end{array}$$

Confluence de la βη-réduction :

- la β-réduction est confluente sur tous les termes (bien typés ou non) du λ-calcul simplement typé,
- pour la η-réduction, il faut adapter la démonstration de la confluence du λ-calcul pur en considérant uniquement les termes bien typés.
  Ex : le terme $\lambda x : A,\ (\lambda y : B,\ y\ x)$ se β-réduit en $\lambda x : A,\ x$ et se η-réduit en $\lambda y : B,\ y$, et il n'existe pas de terme u tel que $\lambda x : A,\ x \rhd^*_{\beta\eta} u$ et $\lambda y : B,\ y \rhd^*_{\beta\eta} u$ (si $A \neq B$).

Normalisation

Le processus qui consiste à appliquer des β-réductions termine-t-il ?
Terme normal : c'est un terme sans radical.

Proposition

Un terme normal est de la forme $t = \lambda x_1 : A_1, \ldots \lambda x_n : A_n,\ (x\ u_1\ \ldots\ u_p)$ où x est une variable et les u_i sont normaux.

Preuve : t est de la forme $\lambda x_1 : A_1, \ldots \lambda x_n : A_n,\ (t'\ u_1\ \ldots\ u_p)$ où t' n'est ni une abstraction ni une application □

- Un terme t est dit faiblement normalisable s'il existe un terme u normal tel que $t \triangleright^* u$.
- Un terme t est dit fortement normalisable si toute suite de réductions issue de t est finie.

En λ-calcul pur il existe des termes faiblement mais non fortement normalisables.

Normalisation

Le processus qui consiste à appliquer des β-réductions termine-t-il ?
Terme normal : c'est un terme sans radical.

Proposition

Un terme normal est de la forme $t = \lambda x_1 : A_1, \dots \lambda x_n : A_n,\ (x\ u_1\ \dots\ u_p)$ où x est une variable et les u_i sont normaux.

Preuve : t est de la forme $\lambda x_1 : A_1, \dots \lambda x_n : A_n,\ (t'\ u_1\ \dots\ u_p)$ où t' n'est ni une abstraction ni une application □

- Un terme t est dit faiblement normalisable s'il existe un terme u normal tel que $t \triangleright^* u$.
- Un terme t est dit fortement normalisable si toute suite de réductions issue de t est finie.

En λ-calcul pur il existe des termes faiblement mais non fortement normalisables.
Ex. $\lambda x\ \lambda y\ y\ (\lambda z\ (z\ z)\ \lambda z\ (z\ z))$

Normalisation forte de la β-réduction

On définit l'ensemble des termes réductibles de type T par induction sur la structure de T :

- si T est atomique alors t est réductible ssi il est fortement normalisable,
- si $T = A \rightarrow B$ alors t est réductible ssi pour tout terme réductible u de type A, $(t\ u)$ est réductible.

Proposition

1. les termes réductibles sont fortement normalisables,
2. les variables sont des termes réductibles.

Preuve Par induction sur la structure de T, type des termes considérés.

- Clair si T est atomique.
- Si $T = A_1 \rightarrow \cdots \rightarrow A_n \rightarrow B$ (B atomique), alors
  (1) soit t un terme réductible de type T, et $x_1, \ldots, x_n$ des variables de types $A_1, \ldots A_n$. Par hypothèse d'induction, $x_1, \ldots, x_n$ sont réductibles, le terme $(t\ x_1\ \ldots\ x_n)$ est donc réductible. Son type est atomique, il est donc fortement normalisable.

Preuve (suite)

- Le terme t est donc aussi fortement normalisable, car si $t_1, t_2, \ldots$ est une suite de réductions issue de t, alors $(t_1\ x_1\ \ldots\ x_n), (t_2\ x_1\ \ldots\ x_n), \ldots$ est une suite de réductions issue de $(t\ x_1\ \ldots\ x_n)$: elle est donc finie.
  (2) Soit x une variable de type T et $u_1, \ldots, u_n$ des termes réductibles de types $A_1, \ldots A_n$. Par hypothèse d'induction, les termes $u_1, \ldots, u_n$ sont fortement normalisables. Toute suite de réductions issue du terme $(x\ u_1\ \ldots\ u_n)$ réduit des radicaux dans les termes $u_1, \ldots, u_n$, elle est donc finie. Le terme $(x\ u_1\ \ldots\ u_n)$ est donc fortement normalisable et de type atomique, il est donc réductible. La variable x est donc un terme réductible □.

Proposition

Tout terme est réductible.

Preuve : Par induction sur la structure du terme t, on montre que si $u_1, \ldots, u_n$ sont des termes réductibles alors $t[x_1 \leftarrow u_1, \ldots, x_n \leftarrow u_n]$ est réductible.
Exercice □

Corollaire

Tout terme est fortement normalisable.

Normalisation forte de la βη-réduction

Proposition

Tout terme est fortement normalisable pour la βη-réduction.

Preuve : Similaire à celle pour la β-réduction □

Corollaire

Tout terme se réduit sur un terme normal unique.

Corollaire

Deux termes sont équivalents s'ils ont la même forme normale.

Corollaire

L'équivalence entre deux termes est décidable.

Typage et déduction naturelle

$$(ax) \quad \frac{(t : A) \in \Gamma}{\Gamma \vdash t : A} \quad (ax)$$

$$(\text{intro}_{\Rightarrow}) \quad \frac{\Gamma[x : A] \vdash t : B}{\Gamma \vdash \lambda x : A,\ t : A \rightarrow B} \quad (\text{abstraction})$$

$$(\text{élim}_{\Rightarrow}) \quad \frac{\Gamma \vdash t : A \rightarrow B \quad \Gamma \vdash u : A}{\Gamma \vdash (t\ u) : B} \quad (\text{application})$$

Exemple : une preuve de

$$(A \Rightarrow B \Rightarrow C) \Rightarrow (A \Rightarrow B) \Rightarrow A \Rightarrow C$$

est

Typage et déduction naturelle

$$(ax) \quad \frac{(t : A) \in \Gamma}{\Gamma \vdash t : A} \quad (ax)$$

$$(\text{intro}_{\Rightarrow}) \quad \frac{\Gamma[x : A] \vdash t : B}{\Gamma \vdash \lambda x : A,\ t : A \to B} \quad (\text{abstraction})$$

$$(\text{élim}_{\Rightarrow}) \quad \frac{\Gamma \vdash t : A \to B \quad \Gamma \vdash u : A}{\Gamma \vdash (t\ u) : B} \quad (\text{application})$$

Exemple : une preuve de

$$(A \Rightarrow B \Rightarrow C) \Rightarrow (A \Rightarrow B) \Rightarrow A \Rightarrow C$$

est

$$\lambda x : (A \to B \to C),\ \lambda y : (A \to B),\ \lambda z : A,\ (x\ z\ (y\ z))$$

Réductions en Coq

```
Require Import Arith.
Require Import ZArith.
Open Scope Z_scope.
Section sqr.

Definition Zsqr (z:Z) : Z := z*z.

Definition my_fun (f : Z -> Z)(z:Z) := f (f z).

Eval cbv delta [my_fun Zsqr] in (my_fun Zsqr).

Eval cbv delta [my_fun] in (my_fun Zsqr).

Eval cbv beta delta [my_fun] in (my_fun Zsqr).
```

Autres réductions : ζ (pour le let), ι (pour les types inductifs)

`Eval compute` est un raccourci pour `Eval cbv iota zeta beta delta`.

Allons plus loin

Le λ-calcul simplement typé n'est pas assez expressif.

- Pas de récursivité.
- Types trop pauvres
  Ex. type des tableaux (taille n) : un type de base pour chaque valeur de n

Une solution : ajouter la récursion et (un peu de) polymorphisme : ML satisfaisant pour la programmation, mais pas pour la vérification

Enrichir les types

λ-calcul simplement typé : les termes et les types ne vivent pas dans le même monde, et les termes sont construits uniquement à partir de termes.

Trois directions : on va utiliser le même langage pour écrire termes et types, et construire

- des types qui dépendent de termes,
- des termes qui dépendent de types,
- des types qui dépendent de types.

Plan

1. Introduction
2. λ-calcul simplement typé
3. Types dépendants
4. Polymorphisme
5. Constructeurs de types
6. Conclusion

Des types qui dépendent de termes

Exemple : tableaux d'entiers

Une famille de types $(tab\ 0)$, $(tab\ 1)$, $\ldots$ $(tab\ n)$, $\ldots$. Considérons la fonction f :

$$(f\ 0) = []$$

$$(f\ 1) = [0]$$

$$(f\ 2) = [0, 0]$$

$$\ldots$$

Le type du résultat de f est $(tab\ n)$ où n est l'argument. Type de f ?

Des types qui dépendent de termes

Exemple : tableaux d'entiers

Une famille de types $(tab\ 0)$, $(tab\ 1)$, $\ldots (tab\ n), \ldots$. Considérons la fonction f :

$$(f\ 0) = []$$

$$(f\ 1) = [0]$$

$$(f\ 2) = [0, 0]$$

$$\ldots$$

Le type du résultat de f est $(tab\ n)$ où n est l'argument. Type de f ?

$$nat \ \rightarrow (tab\ n)\ ?$$

Comment exprimer que n est « le » nat ?

Des types qui dépendent de termes

Exemple : tableaux d'entiers

Une famille de types $(tab\ 0)$, $(tab\ 1)$, ... $(tab\ n)$, ... Considérons la fonction f :

$$(f\ 0) = []$$

$$(f\ 1) = [0]$$

$$(f\ 2) = [0, 0]$$

$$\ldots$$

Le type du résultat de f est $(tab\ n)$ où n est l'argument. Type de f ?

$$nat\ \rightarrow (tab\ n)\ ?$$

Comment exprimer que n est « le » nat ?

$$\Pi n : nat\ (tab\ n)$$

Cas particulier : $A \rightarrow B$ est une notation pour

$$\Pi x : A\ B$$

lorsque x n'apparaît pas dans B.

Les types sont des termes

$\lambda\Pi$-calcul : types et termes dans le même langage.

Les types sont des termes

λΠ-calcul : types et termes dans le même langage.

- Bien former les types : $\Pi n : nat\ (tab\ n)$ mais pas $(tab\ 0\ 0)$.

Les types sont des termes

$\lambda\Pi$-calcul : types et termes dans le même langage.

- Bien former les types : $\Pi n : nat\ (tab\ n)$ mais pas $(tab\ 0\ 0)$.
- Typer les types : $\Pi n : nat\ (tab\ n) : Type$.

Les types sont des termes

λΠ-calcul : types et termes dans le même langage.

- Bien former les types : $\Pi n : nat\ (tab\ n)$ mais pas $(tab\ 0\ 0)$.
- Typer les types : $\Pi n : nat\ (tab\ n) : Type$.
- Typer *Type* :

Les types sont des termes

λΠ-calcul : types et termes dans le même langage.

- Bien former les types : $\Pi n : nat\ (tab\ n)$ mais pas $(tab\ 0\ 0)$.
- Typer les types : $\Pi n : nat\ (tab\ n) : Type$.
- Typer $Type : Type : Type$

Les types sont des termes

$\lambda\Pi$-calcul : types et termes dans le même langage.

- Bien former les types : $\Pi n : nat\ (tab\ n)$ mais pas $(tab\ 0\ 0)$.
- Typer les types : $\Pi n : nat\ (tab\ n) : Type$.
- Typer $Type$: $Type : Kind$

Les types sont des termes

$\lambda\Pi$-calcul : types et termes dans le même langage.

- Bien former les types : $\Pi n : nat\ (tab\ n)$ mais pas $(tab\ 0\ 0)$.
- Typer les types : $\Pi n : nat\ (tab\ n) : Type$.
- Typer $Type$: $Type : Kind$

Quatre catégories de termes :

- $Kind$,
- les genres : $Type$, $nat \to Type, \ldots$ dont le type est $Kind$,
- les types ou familles de types : nat, $(tab\ 0)$, $tab \ldots$, dont le type est un genre,
- les objets : 0, [0] …, dont le type est un type.

λΠ-calcul : contextes bien formés

$$\frac{}{[]\ \text{bien formé}}$$

Déclaration d'une variable de type ou de famille de types :

$$\frac{\Gamma \vdash T : \mathit{Kind}}{\Gamma, x : T\ \text{bien formé}}$$

Déclaration d'une variable d'objet :

$$\frac{\Gamma \vdash T : \mathit{Type}}{\Gamma, x : T\ \text{bien formé}}$$

Type est un genre :

$$\frac{\Gamma\ \text{bien formé}}{\Gamma \vdash \mathit{Type} : \mathit{Kind}}$$

Les variables sont des termes :

$$\frac{\Gamma\ \text{bien formé} \quad x : T \in \Gamma}{\Gamma \vdash x : T}$$

λΠ-calcul : produit et application

Produit (genres) :

$$\frac{\Gamma \vdash T : \mathit{Type} \quad \Gamma, x : T \vdash T' : \mathit{Kind}}{\Gamma \vdash \Pi x : T\ T' : \mathit{Kind}}$$

Produit (types) :

$$\frac{\Gamma \vdash T : \mathit{Type} \quad \Gamma, x : T \vdash T' : \mathit{Type}}{\Gamma \vdash \Pi x : T\ T' : \mathit{Type}}$$

Application :

$$\frac{\Gamma \vdash t : \Pi x : T\ T' \quad \Gamma \vdash t' : T}{\Gamma \vdash (t\ t') : T'[x \leftarrow t']}$$

Rem : on peut former le genre $\mathit{nat} \rightarrow \mathit{Type}$ mais pas le genre $\mathit{Type} \rightarrow \mathit{Type}$.

λΠ-calcul : abstraction et conversion

Abstraction (familles de types)

$$\frac{\Gamma \vdash T : Type \quad \Gamma, x : T \vdash T' : Kind \quad \Gamma, x : T \vdash t : T'}{\Gamma \vdash \lambda x : T,\ t : \Pi x : T\ T'}$$

Abstraction (objets)

$$\frac{\Gamma \vdash T : Type \quad \Gamma, x : T \vdash T' : Type \quad \Gamma, x : T \vdash t : T'}{\Gamma \vdash \lambda x : T,\ t : \Pi x : T\ T'}$$

Conversion

$$\frac{\Gamma \vdash t : T \quad \Gamma \vdash T : Type \quad \Gamma \vdash T' : Type \quad T \equiv T'}{\Gamma \vdash t : T'}$$

Utilisation de la conversion : $(tab'\ n)$ type des tableaux à $n+1$ éléments :

λΠ-calcul : abstraction et conversion

Abstraction (familles de types)

$$\frac{\Gamma \vdash T : Type \quad \Gamma, x : T \vdash T' : Kind \quad \Gamma, x : T \vdash t : T'}{\Gamma \vdash \lambda x : T,\ t : \Pi x : T\ T'}$$

Abstraction (objets)

$$\frac{\Gamma \vdash T : Type \quad \Gamma, x : T \vdash T' : Type \quad \Gamma, x : T \vdash t : T'}{\Gamma \vdash \lambda x : T,\ t : \Pi x : T\ T'}$$

Conversion

$$\frac{\Gamma \vdash t : T \quad \Gamma \vdash T : Type \quad \Gamma \vdash T' : Type \quad T \equiv T'}{\Gamma \vdash t : T'}$$

Utilisation de la conversion : $(tab'\ n)$ type des tableaux à $n+1$ éléments :

$$tab' = \lambda n : nat,\ (tab(S\ n))$$

$$(tab'0) =$$

λΠ-calcul : abstraction et conversion

Abstraction (familles de types)

$$\frac{\Gamma \vdash T : Type \quad \Gamma, x : T \vdash T' : Kind \quad \Gamma, x : T \vdash t : T'}{\Gamma \vdash \lambda x : T,\ t : \Pi x : T\ T'}$$

Abstraction (objets)

$$\frac{\Gamma \vdash T : Type \quad \Gamma, x : T \vdash T' : Type \quad \Gamma, x : T \vdash t : T'}{\Gamma \vdash \lambda x : T,\ t : \Pi x : T\ T'}$$

Conversion

$$\frac{\Gamma \vdash t : T \quad \Gamma \vdash T : Type \quad \Gamma \vdash T' : Type \quad T \equiv T'}{\Gamma \vdash t : T'}$$

Utilisation de la conversion : $(tab'\ n)$ type des tableaux à $n+1$ éléments :

$$tab' = \lambda n : nat,\ (tab(S\ n))$$

$$(tab' 0) = (\lambda n : nat,\ (tab(S\ n))\ 0)$$

qu'on veut convertir en $(tab(S\ 0))$.

Questions

Pour chacun des termes suivants, quelle est sa catégorie, quel est son type ?

- $nat \longrightarrow Type$
- $nat \longrightarrow nat$
- $Kind$
- $\lambda x : nat,\ x$
- $\lambda n : nat,\ (tab(S\ n))$
- $\Pi n : nat\ (tab\ n)$

Termes purs typables

Quels sont les termes purs du λ-calcul que l'on peut typer dans le λΠ-calcul ?

Proposition

Si $\Gamma \vdash t : T$ et t est un objet (*i.e.*, $\Gamma \vdash T : Type$), alors t est une variable, une application d'un objet à un objet ou l'abstraction d'un objet.

Si t est un objet, on définit son contenu comme le terme du λ-calcul pur obtenu à partir de t en éliminant les informations de type. Un terme pur u est dit typable s'il existe un objet typable dans le λΠ-calcul de contenu t.

Proposition

Les termes purs typables dans le λΠ-calcul sont exactement les termes typables du λ-calcul simplement typé.

Preuve : On « aplatit » les types du λΠ-calcul avec un seul type de base et des types flèche □

Normalisation et confluence

D'après la proposition précédente, les termes purs typables dans le $\lambda\Pi$-calcul sont fortement normalisables. La normalisation des termes typés en général nécessite davantage de réductions. Ex. : $\lambda x : (tab\ (\lambda y : nat,\ y\ 0)), x$ se réduit en $\lambda x : (tab\ 0), x$, il n'est pas normal, alors que son contenu $\lambda x\ x$ l'est.
On parvient néanmoins à traduire les termes du $\lambda\Pi$-calcul dans le λ-calcul simplement typé, pour obtenir :

Proposition

Tout terme typé dans le $\lambda\Pi$-calcul est fortement normalisable.

On a également :

Proposition

La $\beta\eta$-réduction du $\lambda\Pi$-calcul est confluente.

Décidabilité du typage

- L'algorithme de calcul du type d'un terme du λ-calcul simplement typé utilise le fait que pour chaque terme, une seule règle de typage peut s'appliquer.
- En $\lambda\Pi$-calcul il n'y a plus unicité du type, à cause de la règle de conversion.
- La décidabilité du typage repose donc sur la décidabilité de l'équivalence entre deux types.
- Cette décidabilité est assurée par les propriétés de normalisation et de confluence.

Déduction naturelle

Le « Π » correspond au « $\forall$ ».

$$(\text{intro}_\forall) \quad \frac{\Gamma \vdash T : \mathit{Type} \quad \Gamma, x : T \vdash T' : \mathit{Type} \quad \Gamma, x : T \vdash t : T'}{\Gamma \vdash \lambda x : T,\ t : \Pi x : T\ T'} \quad (\text{abstraction})$$

$$(\text{élim}_\forall) \quad \frac{\Gamma \vdash t : \Pi x : T\ T' \quad \Gamma \vdash t' : T}{\Gamma \vdash (t\ t') : T'[x \leftarrow t']} \quad (\text{application})$$

Types dépendants en Coq

Changement de notation...

- $\lambda x : T,\ u \longrightarrow$ `fun x : T => u`
- $\Pi x : T\ T' \longrightarrow$ `forall x : T, T'`
- $Type \longrightarrow$ `Set` et `Prop`
- $Kind \longrightarrow$ `Type` (en fait $Type_0, Type_1, \ldots$)

Exemple

```
Parameter binary_word : nat -> Set.

Definition short : Set := binary_word 32.
Definition long : Set := binary_word 64.
```

Exercice

- En déclarant l'existence d'un type `vect` des vecteurs, écrire une fonction qui prend en argument un entier n et renvoie le type des vecteurs de taille n.
- Donner l'arbre de dérivation de $(nat \rightarrow nat) \rightarrow$`Prop`.

Prédicats

$$\frac{\Gamma \vdash T : \texttt{Set} \quad \Gamma, x : T \vdash \texttt{Prop} : \texttt{Type}}{\Gamma \vdash \Pi x : T\ \texttt{Prop} : \texttt{Type}}$$

Un prédicat est un terme de type `A1 -> A2...-> Prop` avec `Ai : Set`
Exemple : `divides : nat -> nat -> Prop`

Question : quel serait le type d'un prédicat exprimant la correction d'un programme triant des listes d'éléments de `Z` ?

Quantification universelle

$$\frac{\Gamma \vdash T : \texttt{Set} \quad \Gamma, x : T \vdash T' : \texttt{Prop}}{\Gamma \vdash \Pi x : T\ T' : \texttt{Prop}}$$

Exemple : `Theorem le_i_ssi : forall i : nat, i <= S (S i)`

Exercice : vérifier que les théorèmes suivants sont bien formés, et construire des termes les habitant.

```
Section A.
  Variables (A:Set)(P Q:A->Prop)(R:A->A->Prop).

  Theorem all_perm : (forall a b:A, R a b)
                         -> forall a b:A, R b a.
  Theorem all_imp_dist : (forall a:A, P a -> Q a)
                               ->(forall a:A, P a)
                                 -> forall a:A, Q a.
End A.
```

Types de données dépendants

$$\frac{\Gamma \vdash T : \texttt{Set} \quad \Gamma, x : T \vdash \texttt{Set} : \texttt{Type}}{\Gamma \vdash \Pi x : T\ \texttt{Set} : \texttt{Type}}$$

Exercice :

- Donner la construction du type de `binary_word`.
- Donner un type de la fonction de concaténation de deux mots binaires de longueurs quelconques.

Plan

1. Introduction
2. λ-calcul simplement typé
3. Types dépendants
4. Polymorphisme
5. Constructeurs de types
6. Conclusion

Polymorphisme : des termes qui dépendent de types

Les règles de formation du produit (genres et types) permettent de former les types

- $nat \to Type$, utile pour définir la famille de types *tab*,
- $\Pi n : nat\ (tab\ n)$ utile pour typer la fonction qui pour tout entier n, renvoie un tableau de taille n rempli de 0.

Mais on ne peut pas construire (et typer) une fonction qui reçoit un type en paramètre et produit un résultat dont le type en dépend.
Ex. $\Pi T : Type\ (T \to T)$, type de la fonction qui, étant donné un type, produit l'identité sur ce type.

On ajoute donc la *règle des types polymorphes*

$$\frac{\Gamma \vdash T : Kind \quad \Gamma, x : T \vdash T' : Type}{\Gamma \vdash \Pi x : T\ T' : Type}$$

et les règles correspondantes pour les abstractions.

Polymorphisme en Coq

On peut maintenant définir la fonction identité polymorphe
```
fun x : Set => (fun a : x => a)
```
qui est de type
```
forall x : Set, x -> x
```

Itérations : le polymorphisme augmente la puissance calculatoire du langage : on peut définir une fonction
```
iterate : forall A : Set, (A->A)->nat->A->A
```
qui va permettre de construire des fonctions non primitives récursives. Par exemple :

```
Definition my_plus : nat -> nat -> nat := iterate nat S.
Definition my_mult (n p:nat) : nat :=
    iterate nat (my_plus n) p 0.
```

Itérations : exercice

- Définir la fonction puissance.
- Définir la fonction d'Ackermann :

$$
\begin{array}{rcl}
Ack(0,n) & = & n+1 \\
Ack(m+1,0) & = & Ack(m,1) \\
Ack(m+1,n+1) & = & Ack(m, Ack(m+1,n))
\end{array}
$$

Plan

1. Introduction
2. λ-calcul simplement typé
3. Types dépendants
4. Polymorphisme
5. Constructeurs de types
6. Conclusion

Constructeurs de types

Il ne reste plus qu'à construire des types qui dépendent de types : on ajoute la *règle des constructeurs de types*

$$\frac{\Gamma \vdash T : \textit{Kind} \quad \Gamma, x : T \vdash T' : \textit{Kind}}{\Gamma \vdash \Pi x : T\ T' : \textit{Kind}}$$

et les règles correspondantes pour les abstractions.

- Cette règle permet par exemple de définir `list`, qui prend en argument un type A, et renvoie le type des listes d'éléments de type A.
- On peut donc construire de nouveaux types à partir de types existants.
- On parle aussi de *types d'ordre supérieur*.

Constructeurs de types en Coq

Exemple d'utilisation : connecteurs propositionnels
Avec cette nouvelle règle, on peut construire les types `Prop->Prop` et `Prop->Prop->Prop`, eux-mêmes de type `Type` ($Type_k$).
Conjonction :

```
Check conj.
conj
     : forall A B : Prop, A -> B -> A /\ B
```

Exercice : donner un terme de preuve du théorème

```
Theorem conj3 : forall P Q R : Prop, P->Q->R->P/\Q/\R.
```

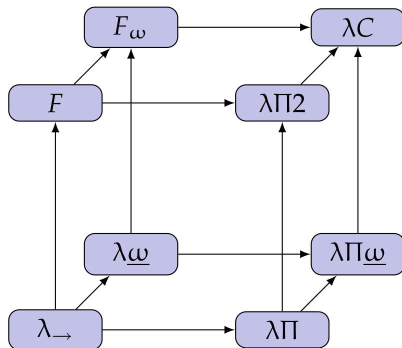
Plan

1. Introduction
2. λ-calcul simplement typé
3. Types dépendants
4. Polymorphisme
5. Constructeurs de types
6. Conclusion

Calcul des Constructions

- Toutes les règles de formation de produits dépendents vues ici.
- Coq = CC + types inductifs.
- Trois directions pour enrichir les types $\rightarrow$ cube de Barendregt
- Tous ces systèmes ont la propriété de normalisation forte.

Les 8 systèmes du cube



- $\uparrow$ termes dépendant de types

$$\frac{\Gamma \vdash T : \mathit{Kind} \quad \Gamma, x : T \vdash T' : \mathit{Type}}{\Gamma \vdash \Pi x : T\ T' : \mathit{Type}}$$

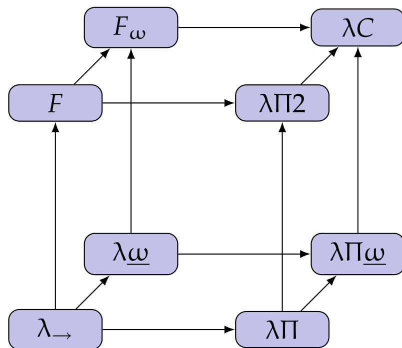
- $\rightarrow$ types dépendant de termes

$$\frac{\Gamma \vdash T : \mathit{Type} \quad \Gamma, x : T \vdash T' : \mathit{Kind}}{\Gamma \vdash \Pi x : T\ T' : \mathit{Kind}}$$

- $\nearrow$ types dépendant de types

$$\frac{\Gamma \vdash T : \mathit{Kind} \quad \Gamma, x : T \vdash T' : \mathit{Kind}}{\Gamma \vdash \Pi x : T\ T' : \mathit{Kind}}$$

Les 8 systèmes du cube



$\uparrow$ termes dépendant de types
$\to$ types dépendant de termes
$\nearrow$ types dépendant de types

- $\lambda_\to$: λ-calcul simplement typé
- F : λ-calcul polymorphe du second ordre
- F_ω : λ-calcul polymorphe d'ordre supérieur
- $\lambda\Pi$: λ-calcul à type dépendent (LF)
- λC : calcul des constructions

Bibliographie

- Barendregt & Geuvers, *Proof Assistants using Dependent Type Systems*
- Dowek, *Théories des types* (poly en ligne)
- le Coq'Art...