Théorie des types en bref

Yves Bertot

Juin 2012

Rappel: le λ-calcul

- Modèle réduit de langage de programmation,
- Extrême simplicité,
  - trois constructions syntaxiques,
    - déclaration de fonction, appel de fonction, valeur de variable
  - une seule entrée acceptée avant de commencer les calculs,
  - une seule sortie retournée après les calculs,
- complications évitées,
  - ordre supérieur: les programmes sont des valeurs,
  - pas de contrôle sur la mémoire nécessaire,
  - stratégies d'exécution variables.

Syntaxe du λ-calcul

- $\lambda x.e$ est la notation pour la fonction qui à x associe e
- on applique les fonctions à des arguments en mettant la fonction à gauche
- $a\ e_1\ e_2 = (a\ e_1)\ e_2$,
  - fonctions à plusieurs arguments comme cas particulier
- si *plus* est la fonction d'addition et $\boxed{1}$ et $\boxed{2}$ sont des nombres, alors *plus* $\boxed{1}$ $\boxed{2}$ est un nombre, et *plus* $\boxed{1}$ est une fonction
- notation $\lambda xy.e$ pour $\lambda x.\lambda y.e$,
- on peut "modéliser" les nombres, les paires, les listes de données,

Calcul avec des λ

- la valeur de $(\lambda x.e)\ a$ est la même que la valeur de e dans laquelle toutes les occurrences de x sont remplacées par a,
- exemple: $(\lambda x.\mathit{plus}\ 1\ x)\ 2 = \mathit{plus}\ 1\ 2$,
- attention aux variables liées: ce sont les occurrences libres de x qui sont remplacées (le résultat ne dépend pas de l'ordre)

$$\begin{aligned}(\lambda x.\ \mathit{plus}((\lambda x.x)\ 1)\ x)\ 2 &= (\lambda x.\ \mathit{plus}\ 1\ x)\ 2\\ (\lambda x.\ \mathit{plus}((\lambda x.x)\ 1)\ x)\ 2 &= \mathit{plus}\ ((\lambda x.x)\ 1)\ 2\end{aligned}$$

- les occurrences de x dans la partie e de $\lambda x.e$ sont appelées occurrences liées,
- les occurrences libres de x dans e sont liées dans $\lambda x.e$.

récursion et calcul infini

- un programme récursif peut s'appeler lui-même
- Exemple $x! = 1$ (si $x = 0$) ou bien $x! = x * (x - 1)!$
- en d'autre termes, il existe une fonction F telle que $f = F\ f$
- Pour *fact*,
  $fact = \lambda x.$ if $x = 0$ then 1 else $x * fact(x - 1)$
  fact est point fixe de
  $\lambda f\ x.$ if $x = 0$ then 1 else $x * f(x - 1)$
- En λ-calcul pur, il existe $Y_T = (\lambda xy.y(xxy))\lambda xy.y(xxy)$, tel que $Y\ F = F(Y\ F)$
- Y permet de représenter des fonctions récursives
- attention aux stratégies de calcul en présence de Y_T
  $Y_T\ F \rightarrow F(Y_T\ F) \rightarrow F(F(Y_T\ F)) \rightarrow \cdots$

Détail du calcul de point fixe

Posons $\theta = (\lambda xy.y(xxy))$

$$\begin{aligned} \theta\theta F &= (\lambda xy.y(xxy))\theta F \\ &= (\lambda y.y(\theta\theta y))F \\ &= (\lambda y.y(\theta\theta y))F \\ &= F(\theta\theta F) \end{aligned}$$

Théorèmes usuels

Church Rosser property si $t \xrightarrow{*} t_1$ et $t \xrightarrow{*} t_2$, alors il existe t_3 t.q. $t_1 \xrightarrow{*} t_3$ et $t_2 \xrightarrow{*} t_3$,

Unicité de la forme normale si $t \xrightarrow{*} t'$ et t' ne peut plus être réduit, alors t' ne dépend pas de la stratégie employée,

Réduction standard La stratégie "extérieur" et à gauche permet d'atteindre la forme normale, lorsqu'elle existe,

- Attention certains termes n'ont pas de forme normale $(\lambda x.xx)\lambda x.xx \rightarrow (\lambda x.xx)\lambda x.xx \rightarrow \cdots$.

Exemple de représentation de données

- booléens : on représente T par $\lambda xy.x$, F par $\lambda xy.y$, If par $\lambda bxy.b\ x\ y$,
- paires P: $\lambda xyz.z\ x\ y$, et projecteurs π_i: $\lambda p.p\ (\lambda x_1\ x_2.x_i)$,
- nombres de Church: n est représenté par $\lambda fx.\overbrace{f(\cdots f}^{n}\ x)\cdots)$,
- addition: $\lambda nm.\lambda fx.n\ f\ (m\ f\ x)$,
  multiplication: $\lambda nm.\lambda f.n\ (m\ f)$,
- test à 0 (appelons-le Q): $\lambda n.n\ (\lambda x.F)\ T$,
- prédécesseur: $\lambda n.\pi_1(n\ (\lambda p.\ P\ (\pi_2\ p)(add\ 1\ (\pi_2\ p)))(P\ 0\ 0))$,
- factorielle: $Y_T\lambda fx.If\ (Q\ x)\ 1\ (mult\ x\ (f\ (pred\ x)))$.

λ-calcul simplement typé

- annoter les fonctions avec des informations sur leurs entrées,
  - Documenter les programmes,
- vérifier sans exécuter les programmes la cohérence des annotations,
- les collections utilisées dans les annotations sont des types,
- notation: $\lambda x : t.\ e$,
- types primitifs, `int`, `bool`, ... mais aussi types de fonctions $t_1 \to t_2$ (par convention $t_1 \to (t_2 \to t_3) \equiv t_1 \to t_2 \to t_3$),

Structures de données et opérations primitives

- le typage supporte l'ajout de données et d'opérations primitives,
- assure que les opérations sont appliquées au bonnes données,
- par exemple, nous ajoutons des couples et des projecteurs:

$$\langle e_1, e_2 \rangle \qquad \mathit{fst}\langle e_1, e_2 \rangle \rightsquigarrow e_1$$

- nouveau type pour les couples: $t_1 * t_2$, pour $\mathit{fst} : t_1 * t_2 \longrightarrow t_1$,
- permet aussi l'utilisation d'entiers natifs.

Exemples

$\lambda f : int \to int \to int.\lambda x : int.f\ x\ (f\ x\ x)$ bien typé
$\lambda f : int \to int.\lambda g : int \to int.f\ (g\ (f\ x))$ bien typé si $x : int$,
$\lambda f : int.\lambda x : int \to int.f\ x$ mal typé,
$f\ f$ mal typé, quelque soit le type de f.

Vérification de types

- première étape: choisir des types pour les variables libres,
- assurer que les fonctions sont appliquées à des expressions de bon type,
- effectuer un parcours récursif des termes,
- Cet algorithme se décrit par des règles d'inférence.

Règles de typage

$$\frac{}{\Gamma, x : t \vdash x : t} \ (1) \qquad \frac{\Gamma \vdash x : t \qquad x \neq y}{\Gamma, y : t' \vdash x : t} \ (2)$$

$$\frac{\Gamma \vdash e_1 : t_1 \qquad \Gamma \vdash e_2 : t_2}{\Gamma \vdash \langle e_1, e_2 \rangle : t_1 * t_2} \ (3)$$

$$\frac{\Gamma, x : t \vdash e : t'}{\Gamma \vdash \lambda x : t.\ e : t \to t'} \ (4)$$

$$\frac{\Gamma \vdash e_1 : t \to t' \qquad \Gamma \vdash e_2 : t}{\Gamma \vdash e_1\ e_2 : t'} \ (5)$$

$$\frac{}{\Gamma \vdash \mathit{fst} : t_1 * t_2 \to t_1} \ (6) \qquad \frac{}{\Gamma \vdash \mathit{snd} : t_1 * t_2 \to t_2} \ (7)$$

Interprétation logique

- lire les noms de types primitifs comme des variables propositionnelles,
- lire les types de fonctions $t_1 \longrightarrow t_2$ comme des implications,
- lire les types de couples $t_1 * t_2$ comme des conjonctions (“et”),
- le type d'un terme clos bien typé est *toujours* une tautologie,
  - *isomorphisme de Curry-Howard*, *types-as-propositions*,
- Pour un type t, trouver un terme e ayant ce type, c'est faire une preuve que c'est une tautologie
- attention, toutes les tautologies ne sont pas prouvables
- exemple: $((A \longrightarrow B) \longrightarrow A) \longrightarrow A$ (formule de Peirce).

La tautologie de Peirce

A	B	$A \to B$	$(A \to B) \to A$	$((A \to B) \to A) \to A$
T	T	T	T	T
T	F	F	T	T
F	T	T	F	T
F	F	T	F	T

Typage et logique

- $\lambda x : A * B.\langle snd\ x, fst\ x\rangle$ est une preuve de $A \wedge B \Rightarrow B \wedge A$,
- plusieurs systèmes de démonstration sont basés sur ce principe: Nuprl, Coq, Agda, Epigram,
- un vérificateur de type est un programme simple
- la recherche de preuves est un problème difficile
- la vérification de preuves est facile,
- le λ-calcul typé est aussi un modèle réduit d'outil de vérification de preuves.

Réduction typée

- même procédé de calcul que pour le λ-calcul pur,
- ajout de la règle de calcul pour les projections sur des couples,
- Théorèmes standard:

  subject reduction theorem typage conservé pendant le calcul,
  weak normalization tout terme typé admet une forme normale,
  strong normalization toute chaine de réductions est finie.

Croisée des chemins

- Vers les langages de programmation
  - inférence de types
  - le polymorphisme
  - la récursion générale
- Vers les systèmes de preuve
  - quantification universelle
  - preuve par récurrence
  - garantie de terminaison des calculs

Récursion structurelle

- éviter les calculs infinis, qui sont des valeurs “indéfinies”,
- fournir un calcul récursif pour certains types seulement,
- prolonger la notion de récursion primitive,
- les termes bien typés représentent des formules prouvables
- référence : le système T de Gödel (voir Girard & Lafont & Taylor *Proofs and types*),

Récursion structurelle sur les entiers

- un nouveau type `nat`,
- trois nouvelles constantes:
  - `O : nat` (représente 0)
  - `S : nat → nat` (représente la fonction *successeur*),
  - `rec_nat`
- `rec_nat` est un récurseur, il permet d'obtenir certaines fonctions récursives facilement,
- Exécution par filtrage (`rec_nat` v f est une fonction récursive)
  - `rec_nat` v f `O` $= v$
  - `rec_nat` v f (`S` n) $= f$ n (`rec_nat` v f n)
- En cohérence avec ce comportement, le type de `rec_nat` est:
  - `rec_nat` : $t \to (\texttt{nat} \to t \to t) \to \texttt{nat} \to t$, pour tout type t,
- la terminaison des calculs est encore garantie par le typage.

Exemples de fonctions récursives

- addition: $plus \equiv \lambda xy.\texttt{rec\_nat}\ y\ (\lambda nv.\texttt{S}\ v)\ x$,
- prédécesseur: $pred \equiv \texttt{rec\_nat}\ 0\ (\lambda nv.n)$,
- soustraction: $minus \equiv \lambda xy.\texttt{rec\_nat}\ x\ (\lambda nv.pred\ v)\ y)$,
  - la soustraction est aussi un test de comparaison, $minus\ x\ y = 0$ si $x \leq y$,
- multiplication: $\lambda xy.\texttt{rec\_nat}\ \texttt{0}\ (\lambda nv.plus\ y\ v)$,
- en général, toute fonction dont on sait calculer le nombre maximal d'appels récursifs (par exemple la division),
- même les fonctions récursives non primitives: Ackermann.

Exemple des arbres binaires (si on a le temps)

- Définissons le type `bin` d'arbres binaires,
- Deux constructeurs :
  - `leaf : bin`,
  - `node : nat → bin → bin → bin`,

Exemple des arbres binaires (2)

- Le récurseur est déterminé par les constructeurs, ici:
  - `rec_bin` a trois arguments (2+1), `rec_bin` f_1 f_2 x, est bien typé si le type de f_1 (respectivement f_2) est adapté au filtrage et à la récursion sur `leaf` (respectivement `node`).
  - f_1 est une valeur de type t,
  - f_2 a (3+2) arguments,
    - 3 est le nombre d'arguments de `node`,
    - 2 est le nombre d'arguments de `node` qui sont de type `bin`,
  - les arguments en plus sont les valeurs d'appels récursifs,

$$\texttt{rec\_bin}\ f_1\ f_2\ (\texttt{node}\ n\ t_1\ t_2) = \\ f_2\ n\ t_1\ (\texttt{rec\_bin}\ f_1\ f_2\ t_1)\ t_2\ (\texttt{rec\_bin}\ f_1\ f_2\ t_2)$$

Récurseurs et filtrage

- Ocaml :

```
let rec_nat fun v f x =
    match x with
        O -> v | S p -> f p (rec_nat v f p)
```

Types dépendants: les familles de types

- fonctions dont les valeurs sont des types,
- fonctions "diagonales": chaque valeur est dans un type différent (déterminé par une autre fonction),
- exemple: A_i une séquence de types représentée par la fonction $A : \texttt{nat} \to \texttt{Type}$, alors on peut imaginer une fonction f telle que:
  - $f\ 0$ a le type $A\ 0$,
  - $f\ 1$ a le type $A\ 1$,
  - $f\ 2$ a le type $A\ 2$,
  - et ainsi de suite,
- on note le type de $f : \Pi x : \texttt{nat}.A\ x$.

Produits dépendants

- un couple de $A_1 \times A_2$ permet d'associer une valeur de type A_i à l'index $i \in \{1, 2\}$,
- plus généralement, une collection de valeurs $(a_0, \ldots, a_n, \ldots)$ permet d'associer une valeur de A_i à l'index $i \in \mathbb{N}$,
- une telle collection est dans $A_0 \times A_1 \times \cdots \times A_n \times \cdots = \Pi_{i \in \mathbb{N}} A_i$,
- la notation de produit indexé est donc bien adaptée pour comprendre les fonctions à type dépendant.

Typage du produit dépendant

$$\frac{\Gamma, x : t \vdash e : t'}{\Gamma \vdash \lambda x : t.e : \Pi x : t.\ t'}$$

$$\frac{\Gamma \vdash e_1 : \Pi x : t.t' \qquad \Gamma \vdash e_2 : t}{\Gamma \vdash e_1\ e_2 : t'[e_2/x]}$$

- La notation $A \longrightarrow B$ est un raccourci pour $\Pi x : A.B$ lorsque x n'apparaît pas dans B,
- si $f : \Pi x : \texttt{nat}.A\ x$ alors, $f\ 1 : A\ 1$.

Interprétation logique du produit dépendant

- si B a le type $A \longrightarrow \texttt{Type}$, l'interprétation dans le cadre logique de B est un prédicat,
- si $t : B\ i$, alors est t est une preuve de $B\ i$,
- si $f : \Pi i : A.B\ i$, alors f permet de construire un terme dans $B\ i$ pour tout $i : A$,
- il faut donc lire $\Pi i : A.B\ i$ comme une quantification universelle
  et $f : \Pi i : A.B\ i$ est la preuve d'une quantification universelle,
- dans Coq, on n'écrit jamais $\Pi i : A.B\ i$ mais toujours $\forall i : A, B\ i$.

Construction de preuves

- supposons qu'il existe un prédicat `even` (parité),
- supposons que l'on ait deux théorèmes (des constantes typées)
  - `even0 : even 0`,
  - `even2 : ∀x : nat, even x → even (S (S x))`,
- on peut composer ces théorèmes pour obtenir la preuve qu'un nombre est pair (dès que c'est vrai),
- par exemple: `even2 0 even0 : even (S (S 0))`
  est une preuve que 2 est pair,
- `even2 2 (even2 0 even0) : even 4`
  est une preuve que 4 est pair,
- `even2 4 (even2 2 (even2 0 even0)) : even 6`,
  et ainsi de suite.

Produit dépendant et polymorphisme explicite

- une fonction polymorphe a un type $T[\alpha]$ pour toutes les valeurs possibles de α,
- ceci peut être décrit explicitement en disant que T est une famille de types
  $T : \texttt{Type} \longrightarrow \texttt{Type}$,
- la fonction polymorphe est alors décrite par le type
  $\Pi x : \texttt{Type}.T\ x$ (donc avec un argument supplémentaire),
- par exemple le type des couples $t_1 * t_2$ cache une constante
  $\texttt{prod} : \texttt{Type} \longrightarrow \texttt{Type} \longrightarrow \mathit{Type}$,
- la notation $\langle e_1, e_2 \rangle$ cache une constante
  $\mathit{pair} : \Pi t_1 : \texttt{Type}.\Pi t_2 : \texttt{Type}.t_1 \longrightarrow t_2 \longrightarrow \texttt{prod}\, t_1\, t_2$,
- à cause du polymorphisme explicite la fonction `pair` a maintenant 4 arguments (et `fst` a 3 arguments).

Produit dépendant et récursion

- `rec_nat` est une constante polymorphe,
- `rec_nat` peut aussi être utilisé pour définir des fonctions à types dépendant!
  - la dépendance doit être exprimée par une fonction $P : \texttt{nat} \to \texttt{Type}$,
  - l'argument fourni pour 0 doit appartenir à $P\ 0$,
  - l'argument fourni pour $S\ n$ doit construire une valeur de type $P\ (S\ n)$,
  - la valeur de l'appel récursif sur n doit appartenir à $P\ n$,
- `rec_nat` :
  $\Pi P : \texttt{nat} \to \texttt{Type}.P\ \texttt{O} \to (\Pi n : \texttt{nat}.P\ n \to P\ (\texttt{S}\ n)) \to$
  $\Pi n : \texttt{nat}.P\ n$
- en interprétation logique, c'est le principe de récurrence sur les entiers naturels!

Types inductifs et dépendances

- des familles de types récursifs,
- les éléments de T_i ont des sous-termes dans T_j,
- exemple: les arbres binaires complets:
  - `hleaf : T 0`,
  - `hnode : ` Πn `:nat. A` $\rightarrow$ `T n` $\rightarrow$ `T n` $\rightarrow$ `T (S n)`
- le type de chaque arbre donne une information sur sa hauteur,
- le constructeur `hnode` indique expressément que les deux sous-arbres doivent avoir la même hauteur,
- le récurseur se construit systématiquement, comme pour les autres types.

Propositions inductives

- dans les familles de types inductives certains types peuvent être habités alors que d'autres ne le sont pas,
- exemple, type `even` indexé par `nat`, avec deux constructeurs:
  - `even0: even 0`,
  - `even2: ∀n:nat. even n → even (S (S n))`,
- En interprétation logique, le type du récurseur exprime que even n'est satisfait que par les nombres pairs,
- `even_ind :`

```
∀ P : nat → Prop,
  P 0 →
  (∀ n:nat, even n → P n → P (S (S n)))→
∀ n:nat, even n → P n
```

Le calcul des constructions inductives: le système Coq

- les propriétés inductives sont très puissantes,
- en Coq, elles sont utilisées pour représenter les connecteurs logiques, la quantification existentielle, et l'égalité, à part $\forall$ et $\longrightarrow$
- la formation des produits dépendants est contrainte par des règles précises qui interdisent d'approcher les paradoxes (Russell, Burali-Forti),
- on peut définir une nouvelle propriété en quantifiant sur toutes les propriétés (*imprédicativité*),
- la définition d'un type inductif doit satisfaire des contraintes sur le type des constructeurs,
- les récurseurs associés aux types inductifs sont remplacés par une construction très générale de récursion structurelle,

Utilisation simple de Coq

- on peut utiliser Coq en ignorant la notion de type dépendant,
  - on ne définit que des fonctions simplement typées avec récursion,
  - on n'utilise des quantifications universelles logiques que dans des formules logiques,
  - les seuls types dépendants que l'on utilise sont des propriétés inductives,
  - des tactiques se chargent de composer des termes complexes.
- On peut aussi utiliser la notion de types dépendants pour une programmation plus sûre