Programmer = démontrer?
La correspondance de Curry-Howard aujourd'hui

Dernier cours

Conclusion, réponses aux questions, discussions

Xavier Leroy

Collège de France

2019-01-30

I

Bilan du cours

La correspondance de Curry-Howard

λ-calcul simplement typé	logique intuitionniste
type	proposition
terme (programme)	démonstration
réduction (exécution)	élimination des coupures
$A \to B$	implication
$A \times B$	conjonction
$A + B$	disjonction
type vide, type unité	$\bot$, $\top$

Se distingue de l'approche ≪proposition = programme≫
(λ-calcul de Church 1942, programmation logique).

≪Implémente≫ l'interprétation BHK de la logique intuitionniste.

Les théories des types modernes

Enrichissement des types et des propositions :
polymorphisme, types dépendants, égalité.

Unification syntaxique entre termes et types, contrôlée par des univers.

$\Rightarrow$ Formalismes unifiés pour programmer et raisonner :
la théorie des types de Martin-Löf, le Calcul des Constructions, les *Pure Type Systems*.

La correspondance de Curry-Howard-Martin Löf

Théorie des types	théorie des ensembles	logique intuitionniste
$A : U$	ensemble	proposition
$A : U$	—	type
$x : A$	élément	démonstration
$0, 1$	$\emptyset, \{\emptyset\}$	$\bot, \top$
$A \times B$	produit cartésien	conjonction
$A + B$	union disjointe	disjonction
$A \to B$	ensemble de fonctions	implication
$x : A \vdash B(x)$	famille d'ensembles	prédicat
$x : A \vdash b : B(x)$	famille d'éléments	démonstration sous hyp.
$\Pi x : A.\, B(x)$	produit	quantificateur «pour tout»
$\Sigma x : A.\, B(x)$	somme disjointe	quantificateur «il existe»
$x =_A y$	égalité	égalité
$p : x =_A y$	—	preuve d'égalité

La correspondance de Curry-Howard-Martin Löf-Voevodsky

(Repris de Emily Riehl, exposé au *Vladimir Voevodsky memorial conference*, 2018)

Théorie des types	théorie des ensembles	logique	théorie de l'homotopie
$A : U$	ensemble	proposition	espace
$A : U$	—	type	—
$x : A$	élément	démonstration	point
$0, 1$	$\emptyset, \{\emptyset\}$	$\bot, \top$	$\emptyset, *$
$A \times B$	produit cartésien	conjonction	espace produit
$A + B$	union disjointe	disjonction	coproduit
$A \to B$	ensemble de fonctions	implication	espace de fonctions
$x : A \vdash B(x)$	famille d'ensembles	prédicat	fibration
$x : A \vdash b : B(x)$	famille d'éléments	dém. sous hyp.	section
$\Pi x : A.\, B(x)$	produit	≪pour tout≫	espace des sections
$\Sigma x : A.\, B(x)$	somme disjointe	≪il existe≫	espace total
$x =_A y$	égalité	égalité	espace de chemins
$p : x =_A y$	—	preuve d'égalité	chemin de x à y

Types et prédicats inductifs

Un mécanisme général pour définir

- des types de données engendrés par des constructeurs ;
- des prédicats engendrés par des axiomes et des règles

ainsi que les fonctions récursives et les démonstrations par récurrence associés.

S'étend *mutatis mutandis* à la coinduction et aux codonnées.

Vers la logique classique

De belles correspondances :

- traductions par double négation / traductions par passage de continuations (CPS) ;
- lois classiques / opérateurs de contrôle (`call/cc`).

Une question encore ouverte : quelle est ≪le bon≫ formalisme de calcul pour exprimer le contenu calculatoire d'une démonstration classique ?
(lambda-calculs symétriques, machines à la Krivine, calculs de processus, réseaux d'interaction, etc.)

Transformations de programmes et de démonstrations

Transformer les programmes d'un langage L_1 vers un langage L_2 : une technique standard de compilation, de sémantique, et de programmation.

Transformer les propositions et les démonstrations d'une logique L_1 vers une logique L_2 :

- les traductions par double négation ;
- le forcing intuitionniste ;
- la paramétricité à la manière de Bernardy et al ;
- les modèles syntaxiques de Boulier, Pédrot et Tabareau ;
- etc.

Les effets

Les principaux effets :

- Partialité (récursion générale, non-terminaison).
- Mutabilité (modifications ≪en place≫).
- Exceptions, opérateurs de contrôle.
- Communications : entrées-sorties, parallélisme à mémoire partagée, parallélisme par passage de messages.

Les monades comme représentation de nombreux effets.

Les effets algébriques et les gestionnaires d'effets comme représentation plus souple de certains effets.

Des logiques de programmes pour raisonner sur certains effets (logique de Hoare, logiques de séparation, etc).

Pas de correspondance générale du côté logique.

Quelques outils pour la sémantique

Des outils plus ou moins inspirés par la logique pour raisonner sur les programmes et donner la sémantique de langages de programmation :

- Les relations logiques, indicées par des types ou par des comptes de pas (*step-indexing*).
- Le ≪topos des arbres≫ et sa modalité $\triangleright$, ≪plus tard≫, pour construire des objets sémantiques (et des programmes réactifs !) par récursion gardée.

II

Curry-Howard fait-il de moi un meilleur programmeur ?

La primauté de la programmation fonctionnelle pure et totale

À la base de tout programme il y a une collection de fonctions pures (sans état) et totales (terminant toujours).

Au cœur de tout langage de programmation il devrait y avoir un langage fonctionnel pur, de préférence typé, de préférence garantissant la terminaison.

Quelques raisons :

- Ces fonctions sont à la fois des programmes et des définitions mathématiques, sur lesquelles on peut raisonner directement (sans logique de programme).
- ≪Pur + total≫ permet un typage statique avec des types riches : types dépendants, équations façon HITs, etc.
- ≪Pur + total≫ permet au langage d'exprimer des termes de preuve.

Partialité et récursion générale

Mauvaises raisons :

- ≪Pour être Turing-complet.≫
  (Tous les calculs utiles sont prouvablement terminants.)
- ≪Un serveur Web ne doit pas terminer !≫
  (Si, pour chaque requête $\Rightarrow$ notion de productivité.)

Bonnes raisons :

- Démontrer la terminaison d'un algorithme peut être difficile.
- Coder un algorithme dans un langage normalisant, encore plus.
- Pour beaucoup d'applications, la correction partielle suffit.

Pour aller plus loin :

- Garantir le temps d'exécution dans le pire cas (WCET).
- Garantir une complexité asymptotique.

Programmation impérative et données mutables

Mauvaises raisons :

- ≪Un algorithme, c'est une recette de cuisine !≫
- ≪C'est comme ça que fonctionne le matériel !≫

Bonnes raisons :

- Beaucoup des meilleurs algorithmes connus utilisent de l'état mutable (les algorithmes fonctionnels sont $\log n$ plus lents).
- Programmation système de bas niveau.

Médiation :

- Encapsulation de l'état mutable dans une interface pure.
- Linéarité et contrôle du partage : logiques de séparation, *ownership types*, types comme permissions, le langage Rust.

Objets, classes, héritage, modules, composants, …

Mauvaises raisons :

- ≪La nature est une hiérarchie de classes.≫
- Tout code doit être extensible a posteriori, coûte que coûte.

Bonnes raisons :

- Réutilisation de code et de sa vérification.
- Décomposition modulaire + barrières d'abstraction.
  (Une source d'inspiration : les structures algébriques.)
- Mécanismes de base bien compris : abstraction de fonctions (λ), abstraction de types ($\exists$), polymorphisme paramétrique ($\forall$).

Pour aller plus loin :

- Beaucoup de mécanismes de haut niveau, mal compris, insuffisants.

III

Questions et discussions