Sémantiques mécanisées, septième cours

Des fonctions et des types: la sémantique d'un langage fonctionnel

Xavier Leroy

2020-02-06

Collège de France, chaire de sciences du logiciel

Un changement de paradigme

IMP, un petit langage impératif :

- Exécuter un programme = modifier un état.
- Opération de base : l'affectation.
- Structures de contrôle : conditionelle, boucles.
- Types de données : premier ordre (nombres).

FUN, un petit langage fonctionnel :

- Exécuter un programme = calculer sa valeur.
- Opérations de base : définir, appliquer des fonctions.
- Structures de contrôle : conditionelle, récursion.
- Types de données : ordre supérieur
  (fonctions comme valeurs de première classe).

Le langage fonctionnel FUN

Une recette de langage fonctionnel

$$
\begin{array}{ll}
 & \text{le } \lambda\text{-calcul} \\
+ & \text{une stratégie de réduction} \\
+ & \text{des types de données primitifs} \\
+ & \text{un système de types} \\
\hline
= & \text{un langage fonctionnel}
\end{array}
$$

Le lambda-calcul

Termes : $M, N ::= x$ variables

$\mid \lambda x.\, M$ abstraction de fonction ($x \mapsto M$)

$\mid M\ N$ application de fonction

Une règle structurelle : la α-conversion
(les variables liées peuvent être renommées)

$$\lambda x.\, M =_\alpha \lambda y.\, M\{x \leftarrow y\} \qquad \text{si } y \text{ non libre dans } M$$

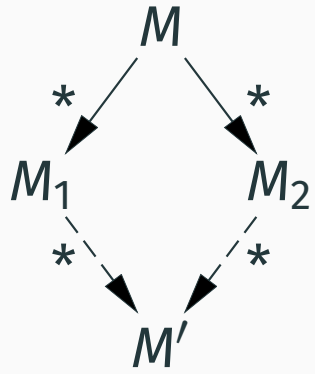
Une règle de calcul : la β-réduction

$$(\lambda x.\, M)\ N \to_\beta M\{x \leftarrow N\}$$

Bonnes propriétés des réductions

Théorème (Church et Rosser, 1935)

La β-réduction est confluente : si $M \xrightarrow{*} M_1$ et $M \xrightarrow{*} M_2$, il existe M' tel que $M_1 \xrightarrow{*} M'$ et $M_2 \xrightarrow{*} M'$.



On dit que N est forme normale de M si $M \xrightarrow{*} N \not\to$

Corollaire

La forme normale d'un terme, si elle existe, est unique.

Expressivité du lambda-calcul

Le lambda-calcul est Turing-complet.

En particulier, il peut exprimer, via des codages fonctionnels,

- Tous les types de données usuels : entiers, paires, listes, …

  Exemple : le codage de Church des entiers

$$n \equiv \lambda f.\, \underbrace{f \circ \cdots \circ f}_{n \text{ fois}} \equiv \lambda f.\, \lambda x.\, \underbrace{f\ (f\ (\cdots\ (f\ x)))}_{n \text{ fois}}$$

- La récursion générale via des combinateurs de point fixe.

  Exemple : le combinateur $Y = \lambda f.\, (\lambda x.\, f\ (x\ x))\ (\lambda x.\, f\ (x\ x))$ qui vérifie $Y\ F \xrightarrow{*} F\ (Y\ F)$.

Pourquoi le lambda-calcul n'est pas un bon langage

Peu de contrôle sur la terminaison et la complexité des calculs.

Non déterminisme des β-réductions, qui peuvent s'appliquer à plusieurs endroits et dans n'importe quel ordre. Suivant la manière d'enchaîner les β-réductions,

- un calcul peut diverger ou terminer ;
- terminer rapidement ou lentement.

Les codages fonctionnels sont limités.

- Peu naturels.
- Généralement inefficaces.
- Souvent non typables dans les systèmes de types usuels.

Stratégies de réduction

Déterminiser la manière dont on applique la β-réduction à chaque étape de calcul. Deux principaux choix :

- **Réduction forte ou faible** : on réduit ou pas ≪sous les λ≫.
  Réduction faible : le corps d'une fonction n'est évalué qu'après application.
  Réduction forte : on peut simplifier le corps d'une fonction avant application.
- **Appel par nom ou appel par valeur** :
  Par valeur : l'argument doit être évalué avant d'être passé à la fonction.
  Par nom : l'argument est passé tel quel (non évalué).

Spécifier une stratégie : le style ≪SOS≫

(G. Plotkin, *A structural approach to operational semantics*, 1981, 2004.)

Des axiomes et des règles pour la relation $M \to M'$
(lire : le terme M tout entier se réduit en le terme M').

Appel par nom faible

$$(\lambda x.\, M)\; N \to M\{x \leftarrow N\}$$

$$\frac{M \to M'}{M\; N \to M'\; N}$$

Appel par valeur faible de gauche à droite

$$(\lambda x.\, M)\; v \to M\{x \leftarrow v\}$$

$$\frac{M \to M'}{M\; N \to M'\; N} \qquad \frac{N \to N'}{v\; N \to v\; N'}$$

(Les valeurs, notées v, sont ici les lambdas : $v ::= \lambda x.\, M$)

Spécifier une stratégie : par une grammaire des contextes

(A. Wright, M. Felleisen, *A Syntactic Approach to Type Soundness*, 1994).

Une règle générique de réduction sous un contexte E :

$$\frac{M \to_\varepsilon M' \qquad E \in Ctx}{E[M] \to E[M']}$$

Pour chaque stratégie, des axiomes pour la réduction en tête $\to_\varepsilon$ et une grammaire des contextes valides E :

Appel par nom faible

$(\lambda x.\, M)\ N \to_\varepsilon M\{x \leftarrow N\}$

$E ::= [\,] \mid E\ N$

Appel par valeur faible

$(\lambda x.\, M)\ v \to_\varepsilon M\{x \leftarrow v\}$

Gauche-droite : $E ::= [\,] \mid E\ N \mid v\ E$

Droite-gauche : $E ::= [\,] \mid E\ v \mid M\ E$

Spécifier une stratégie : par une sémantique naturelle

Comme nous l'avions fait pour IMP, nous pouvons résumer les suites finies de réductions vers une valeur $M \xrightarrow{*} v \not\to$ par un prédicat $M \Downarrow v$, «le terme M s'évalue en la valeur v».

Appel par nom faible :

$$\lambda x.\, M \Downarrow \lambda x.\, M \qquad \frac{M \Downarrow \lambda x.\, P \qquad P\{x \leftarrow N\} \Downarrow v}{M\, N \Downarrow v}$$

Appel par valeur faible :

$$\lambda x.\, M \Downarrow \lambda x.\, M \qquad \frac{M \Downarrow \lambda x.\, P \qquad N \Downarrow v' \qquad P\{x \leftarrow v'\} \Downarrow v}{M\, N \Downarrow v}$$

Ajout de types de données primitifs

Un schéma systématique : ajouter

- des formes syntaxiques à la grammaire des termes ;
- des règles de réduction en tête ;
- des cas de valeurs et de contextes.

Point de départ : l'appel par valeur faible.

Termes : $M, N ::= x \mid \lambda x.\, M \mid M\ N$

Valeurs : $v ::= \lambda x.\, M$

Contextes : $E ::= [\,] \mid E\ M \mid v\ E$

Réduction en tête : $(\lambda x.\, M)\ v \to_\varepsilon M\{x \leftarrow v\}$

Les booléens

Termes : $M ::= \ldots \mid \texttt{true} \mid \texttt{false} \mid \texttt{if}\ M_1\ M_2\ M_3$

Valeurs : $v ::= \ldots \mid \texttt{true} \mid \texttt{false}$

Contextes : $E ::= \ldots \mid \texttt{if}\ E\ M_2\ M_3$

$$\texttt{if true}\ M_2\ M_3 \to_\varepsilon M_2$$

$$\texttt{if false}\ M_2\ M_3 \to_\varepsilon M_3$$

Les entiers de Peano

$$\begin{array}{lll} \text{Termes :} & M & ::= \ldots \mid \mathtt{0} \mid S\ M \mid \mathtt{if0}\ M_1\ M_2\ M_3 \\ \text{Valeurs :} & v & ::= \ldots \mid \mathtt{0} \mid S\ v \\ \text{Contextes :} & E & ::= \ldots \mid S\ E \mid \mathtt{if0}\ E\ M_2\ M_3 \end{array}$$

$$\mathtt{if0}\ \mathtt{0}\ M_2\ M_3 \to_\varepsilon M_2$$

$$\mathtt{if0}\ (S\ v)\ M_2\ M_3 \to_\varepsilon M_3\ v$$

Les produits et les sommes

$$
\begin{aligned}
\text{Termes :} \quad & M ::= \ldots \mid (M_1, M_2) \mid \mathtt{fst}\ M \mid \mathtt{snd}\ M \\
& \qquad \mid \mathtt{left}\ M \mid \mathtt{right}\ M \mid \mathtt{case}\ M\ M_1\ M_2 \\
\text{Valeurs :} \quad & v ::= \ldots \mid (v_1, v_2) \mid \mathtt{left}\ v \mid \mathtt{right}\ v \\
\text{Contextes :} \quad & E ::= \ldots \mid (E, M) \mid (v, E) \mid \mathtt{fst}\ E \mid \mathtt{snd}\ E \\
& \qquad \mid \mathtt{left}\ E \mid \mathtt{right}\ E \mid \mathtt{case}\ E\ M_2\ M_3
\end{aligned}
$$

$$
\begin{aligned}
\mathtt{fst}\ (v_1, v_2) &\to_\varepsilon v_1 & \qquad \mathtt{case}\ (\mathtt{left}\ v)\ M_2\ M_3 &\to_\varepsilon M_2\ v \\
\mathtt{snd}\ (v_1, v_2) &\to_\varepsilon v_2 & \qquad \mathtt{case}\ (\mathtt{right}\ v)\ M_2\ M_3 &\to_\varepsilon M_3\ v
\end{aligned}
$$

Les points fixes

Termes : $M ::= \ldots \mid \mathtt{fix}\, M$

Valeurs : $v ::= \ldots \mid \mathtt{fix}\, v$

Contextes : $E ::= \ldots \mid \mathtt{fix}\, E$

$$\mathtt{fix}\, v_f\, v \to_\varepsilon v_f\, (\mathtt{fix}\, v_f)\, v$$

Mécaniser un langage fonctionnel et sa sémantique

Voir le fichier coq `FUN.v`.

Les outils de base sont les mêmes que pour IMP :

- Types inductifs pour la syntaxe abstraite.
- Prédicats inductifs pour les relations de réduction et d'évaluation.

Seul problème embarrassant : la α-conversion

$$\lambda x.\, M =_\alpha \lambda y.\, M\{x \leftarrow y\} \qquad \text{si } y \text{ non libre dans } M$$

Il n'est pas évident de traiter les termes modulo α-conversion, c.à.d. à renommage près des variables liées.

Vivre sans alpha-conversion

Le développement `FUN.v` représente les termes sans renommage implicite des variables liées :

```
Abs("x", Var "x")  ≠  Abs("y", Var "y")
```

Cela pose problème pour définir la substitution $M\{x \leftarrow N\}$:
la définition naïve

$$(\lambda y.\, M)\{x \leftarrow N\} = \lambda y.\, (M\{x \leftarrow N\})$$

est vulnérable aux captures de variables.

P.ex. $(\lambda y.\, x)\{x \leftarrow y\}$ est calculé comme $\lambda y.\, y$ ✘

Vivre sans alpha-conversion

La définition naïve de la substitution

$$(\lambda y.\, M)\{x \leftarrow N\} = \lambda y.\, (M\{x \leftarrow N\})$$

est correcte si le terme N est clos, c.à.d. sans variables libres.

(Si N est clos, $\lambda y \ldots N \ldots$ ne peut pas capturer un y libre dans N.)

Par un heureux hasard, la réduction d'un terme clos (un programme complet p.ex.) ne met en jeu que des termes clos :

$$\underbrace{\mathit{Prog}}_{\text{clos}} \to \cdots \to \underbrace{(\lambda x.\, M)}_{\text{clos}}\ \underbrace{N}_{\text{clos}} \to \underbrace{M\{x \leftarrow N\}}_{\text{clos}} \to \cdots$$

On obtient donc une sémantique correcte pour les programmes complets uniquement …

Un système de types simples

Des programmes absurdes

≪On n'additionne pas des choux et des carottes.≫

≪On ne mélange pas les torchons et les serviettes.≫

Lorsqu'on ajoute au lambda-calcul des types de données, comme p.ex. les booléens, des termes absurdes apparaissent :

- $\texttt{true}\ (\lambda x.\, x)$ (un booléen utilisé comme une fonction)
- $\texttt{if}\ (\lambda x.\, x)\ M\ M'$ (une fonction utilisée comme un booléen)

Typage dynamique, typage statique

Typage dynamique :

détecter et signaler ces absurdités durant l'exécution.

$$(\lambda b.\ \texttt{if}\ b\ M\ M')\ (\lambda x.\ x) \to \texttt{if}\ (\lambda x.\ x)\ M\ M' \to \texttt{ERREUR!}$$

Typage statique :

analyser les termes avant exécution pour rejeter «statiquement» ceux qui ne respectent pas le bon typage.

- ✔ $\lambda b : \texttt{bool}.\ \texttt{if}\ b\ \texttt{false}\ \texttt{true} : \texttt{bool} \to \texttt{bool}$
- ✘ $(\lambda b : \texttt{bool}.\ \texttt{if}\ b\ \texttt{false}\ \texttt{true})\ (\lambda x.\ x)$
- ✘ $\lambda b : \texttt{bool} \to \texttt{bool}.\ \texttt{if}\ b\ \texttt{false}\ \texttt{true}$

Un système de types (statique)

Une algèbre de types, p.ex. les «types simples»

Types : $\tau, \sigma ::= \texttt{bool}$ type de base

$\mid \sigma \to \tau$ type des fonctions de σ dans τ

Des règles de typage définissant la relation $\Gamma \vdash M : \tau$

qui se lit «dans le contexte Γ le terme M a le type τ».

Le contexte Γ est une liste d'hypothèses $x_1 : \tau_1, \ldots, x_n : \tau_n$ qui associe à chaque variable libre x_i son type τ_i.

Les règles de typage pour les types simples

Le lambda-calcul simplement typé :

$$\frac{\Gamma = \ldots, x : \tau, \ldots}{\Gamma \vdash x : \tau}\ \text{(Var)} \qquad \frac{x \notin \mathit{Dom}(\Gamma) \quad \Gamma, x : \sigma \vdash M : \tau}{\Gamma \vdash \lambda x.\, M : \sigma \to \tau}\ \text{(Abs)}$$

$$\frac{\Gamma \vdash M : \sigma \to \tau \quad \Gamma \vdash N : \sigma}{\Gamma \vdash M\ N : \tau}\ \text{(App)}$$

L'extension aux booléens :

$$\Gamma \vdash \texttt{true} : \texttt{bool}\ \text{(Cst)} \qquad \Gamma \vdash \texttt{false} : \texttt{bool}\ \text{(Cst)}$$

$$\frac{\Gamma \vdash M : \texttt{bool} \quad \Gamma \vdash N : \tau \quad \Gamma \vdash P : \tau}{\Gamma \vdash \texttt{if}\ M\ N\ P : \tau}\ \text{(If)}$$

La sûreté d'un système de types (*type soundness*)

Well-typed programs do not go wrong. (R. Milner)

Un système de types est sûr si aucun programme bien typé dans le contexte vide ne peut produire d'erreurs à l'exécution (du genre de `true` ($\lambda x.x$)).

En termes de suites de réductions :

$$\begin{aligned} \text{Terminaison normale :} &\quad M \to \cdots \to v \in \mathit{Val} \\ \text{Terminaison en erreur :} &\quad M \to \cdots \to N \not\to, N \notin \mathit{Val} \\ \text{Divergence :} &\quad M \to \cdots \to M' \to \cdots \end{aligned}$$

Sûreté du typage = si $\emptyset \vdash M : \tau$, le cas «terminaison en erreur» est impossible.

(Normalisation = si $\emptyset \vdash M : \tau$, le cas «divergence» est impossible.)

Diverses manières de démontrer la sûreté

Avec une sémantique dénotationnelle : (1975–1985)

(D. MacQueen, G. Plotkin, R. Sethi, *An ideal model for recursive polymorphic types*, 1986)

- Écrire une sémantique dénotationnelle $[\![M]\!]$ à valeurs dans un domaine avec un point spécial $\mathtt{err}$
  p.ex. $D \simeq \mathit{Bool}_\perp + [D \to D] + \{\mathtt{err}\}_\perp$.
- Interpréter les types τ par des ensembles $[\![\tau]\!]$ ne contenant pas $\mathtt{err}$.
- Montrer que si $\emptyset \vdash M : \tau$, alors $[\![M]\!] \in [\![\tau]\!]$

Diverses manières de démontrer la sûreté

Avec une sémantique dénotationnelle : (1975–1985)

Avec une sémantique naturelle : (1980–1995)

(M. Tofte, *Operational semantics and polymorphic type inference*, PhD Edinburgh, 1988)

- Écrire deux sémantiques naturelles : $M \Downarrow v$ pour la terminaison normale, $M \Downarrow \texttt{err}$ pour la terminaison en erreur.
- Montrer que si $\emptyset \vdash M : \tau$, alors $M \not\Downarrow \texttt{err}$, et $M \Downarrow v \Rightarrow v \in \tau$.

Diverses manières de démontrer la sûreté

Avec une sémantique dénotationnelle : (1975–1985)

Avec une sémantique naturelle : (1980–1995)

Avec une sémantique à réductions : (depuis 1995)

(A. Wright et M. Felleisen, *A syntactic approach to type soundness*, 1994)

- Montrer deux propriétés : progression et préservation.

La propriété de progression

Montrer qu'un programme bien typé ne produit pas immédiatement une erreur.

Théorème (Progression)

Si $\emptyset \vdash M : \tau$ alors ou bien M est une valeur, ou bien M se réduit ($M \to N$ pour un certain N).

Utilise un lemme qui détermine les formes des valeurs suivant leur type.

Lemme (Formes canoniques)

Soit v une valeur.
Si $\emptyset \vdash v : \sigma \to \tau$ alors v est de la forme $\lambda x.\, M$.
Si $\emptyset \vdash v :$ `bool` alors v est `true` *ou* `false`.

La propriété de préservation (*subject reduction*)

Le bon typage est préservé par les étapes de réduction.

Théorème (Préservation)

Si $\Gamma \vdash M : \tau$ *et* $M \to N$, *alors* $\Gamma \vdash N : \tau$.

Utilise un lemme de substitution et un lemme d'affaiblissement.

Lemme (Stabilité du typage par substitution)

Si $\Gamma, x : \sigma, \Gamma' \vdash M : \tau$ *et* $\Gamma \vdash N : \sigma$, *alors* $\Gamma, \Gamma' \vdash M\{x \leftarrow N\} : \tau$.

Lemme (Affaiblissement)

Si $\Gamma \vdash M : \tau$ *alors* $\Gamma, \Gamma' \vdash M : \tau$.

La sûreté du typage

Well-typed programs do not go wrong.

Soit M un programme clos bien typé : $\emptyset \vdash M : \tau$.

Supposons que M termine en erreur :

$$M \to \cdots \to N \not\to, N \notin \mathit{Val}$$

Par préservation (itérée), $\emptyset \vdash N : \tau$.

Par progression, ou bien N est une valeur, ou bien N se réduit.

Contradiction !

Termes intrinsèquement typés

Deux visions du typage

Vision ≪extrinsèque≫, à la manière de Curry :

- Syntaxe abstraite et sémantique sont définies indépendamment du système de types.
- Le système de types est un ≪filtre≫ (une analyse statique) qui élimine certains termes à problèmes.

Vision ≪intrinsèque≫, à la manière de Church :

- Le système de types fait partie de la définition des termes du langage. P.ex. le lambda-calcul simplement typé de Church :

$$M_\tau ::= x_\tau \mid (\lambda x_\sigma .\, M_\tau)_{\sigma\to\tau} \mid (M_{\sigma\to\tau}\, N_\sigma)_\tau$$

- La sémantique est définie uniquement sur les termes typés.

Types dépendants et termes intrinsèquement typés

La vision ≪intrinsèque≫ de Church s'exprime à l'aide de types dépendants (Coq, Agda, …) ou juste de types algébriques généralisés (GADTs) (Haskell, OCaml).

Le type des termes $\texttt{term}\ \Gamma\ \tau$ est paramétré par un contexte Γ et une expression de type τ.

$$\texttt{Const} : \texttt{bool} \to \texttt{term}\ \Gamma\ \texttt{Bool}$$

$$\texttt{Cond} : \texttt{term}\ \Gamma\ \texttt{Bool} \to \texttt{term}\ \Gamma\ \tau \to \texttt{term}\ \Gamma\ \tau \to \texttt{term}\ \Gamma\ \tau$$

$$\texttt{App} : \texttt{term}\ \Gamma\ (\texttt{Fun}\ \sigma\ \tau) \to \texttt{term}\ \Gamma\ \sigma \to \texttt{term}\ \Gamma\ \tau$$

$$\texttt{Abs} : \texttt{term}\ (\sigma :: \Gamma)\ \tau \to \texttt{term}\ \Gamma\ (\texttt{Fun}\ \sigma\ \tau) \qquad (?)$$

$$\texttt{Var} : \texttt{var}\ \Gamma\ \tau \to \texttt{term}\ \Gamma\ \tau \qquad (?)$$

Comment représenter les variables ?

Dans la vision ≪intrinsèque≫, une variable désigne une des entrées du contexte. Cette entrée détermine son type. Il est impossible de parler d'une variable non décrite par le contexte !

Désignation par un nom :

possible, modulo problèmes de renommage.

Désignation par une position :

plus naturelle : contexte $\approx$ liste, entrée $\approx$ position dans la liste.

C'est la notation de de Bruijn (1972) !

La notation de de Bruijn

(N. de Bruijn, *Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation*, 1972.)

Au lieu d'identifier les variables par leurs noms, la notation de de Bruijn les identifie par leur position par-rapport à la λ-abstraction qui les lie.

```
λx. (λy. y x) x
       | |  |
λ.  (λ.  1 2) 1
```

$\underline{n}$ est la variable liée par le n^{e} λ englobant.

Deux termes α-convertibles sont égaux en notation de de Bruijn : $\lambda x.\, x$ et $\lambda y.\, y$ sont représentés par $\lambda.\, \underline{1}$

Notation de de Bruijn intrinsèquement typée

Un contexte Γ est une liste de types $\tau_1 :: \cdots :: \tau_n :: \mathit{nil}$ avec τ_i le type de la variable d'indice i en notation de de Bruijn.

Le type $\mathtt{var}\ \Gamma\ \tau$ des variables de type τ dans le contexte Γ est isomorphe aux entiers compris entre 1 et la taille de Γ.

Il est engendré par deux constructeurs :

$$\begin{array}{lcll} \mathtt{V1} & : & \mathtt{var}\ (\tau :: \Gamma)\ \tau & \text{(unité)} \\ \mathtt{VS} & : & \mathtt{var}\ \Gamma\ \tau \to \mathtt{var}\ (\sigma :: \Gamma)\ \tau & \text{(successeur)} \end{array}$$

Définitions dérivées :

$$\begin{array}{lcll} \mathtt{V2} & = & \mathtt{VS}\ \mathtt{V1} & : & \mathtt{var}\ (\tau_1 :: \tau_2 :: \Gamma)\ \tau_2 \\ \mathtt{V3} & = & \mathtt{VS}\ \mathtt{V2} & : & \mathtt{var}\ (\tau_1 :: \tau_2 :: \tau_3 :: \Gamma)\ \tau_3 \end{array}$$

Sémantique dénotationnelle de termes intrinsèquement typés

On peut définir une interprétation des types de FUN comme des types de Coq :

$$[\![\texttt{Bool}]\!] = \texttt{bool} \qquad [\![\texttt{Fun}\ \sigma\ \tau]\!] = [\![\sigma]\!] \to [\![\tau]\!]$$

Les contextes FUN deviennent les types Coq des environnements d'évaluation qui associent une valeur à chaque variable du contexte :

$$[\![\mathit{nil}]\!] = \texttt{unit} \qquad [\![\tau :: \Gamma]\!] = [\![\tau]\!] * [\![\Gamma]\!]$$

On peut alors interpréter un terme $a : \texttt{term}\ \Gamma\ \tau$ comme une fonction Coq environnement $\mapsto$ valeur :

$$[\![a]\!] : [\![\Gamma]\!] \to [\![\tau]\!]$$

Sémantique dénotationnelle de termes intrinsèquement typés

$$
\begin{aligned}
[\![\texttt{Var V1}]\!]\ e &= \texttt{fst}(e) \\
[\![\texttt{Var (VS } v\texttt{)}]\!]\ e &= [\![\texttt{Var } v]\!]\ (\texttt{snd } e) \\
[\![\texttt{Abs } a]\!]\ e &= \texttt{fun } x \Rightarrow [\![a]\!]\ (x, e) \\
[\![\texttt{App } a_1\ a_2]\!]\ e &= ([\![a_1]\!]\ e)\ ([\![a_2]\!]\ e) \\
[\![\texttt{Const } b]\!]\ e &= b \\
[\![\texttt{Cond } a_1\ a_2\ a_3]\!]\ e &= \texttt{if } [\![a_1]\!]\ e \texttt{ then } [\![a_2]\!]\ e \texttt{ else } [\![a_3]\!]\ e
\end{aligned}
$$

Ceci définit une fonction Coq qui est bien typée et totale
$\Rightarrow$ sûreté du typage et normalisation forte ≪par construction≫.

Satisfait les équations de la sémantique dénotationnelle.

Compatible avec les réductions : si $a \to a'$ alors $[\![a]\!] = [\![a']\!]$.

Limitations de l'approche intrinsèque

Les traits du langage objet (FUN) doivent être disponibles ou encodables dans le langage hôte (Coq).

- Effets (y compris divergence) $\Rightarrow$ codage monadique.
- Sous-typage $\Rightarrow$ coercions $[\![\mathit{soustype}]\!] \to [\![\mathit{supertype}]\!]$.
- Polymorphisme imprédicatif (Système F) $\Rightarrow$ Coq `-impredicative-set`.

Le langage hôte doit avoir des familles inductives (GADTs) et de préférence des types dépendants $\Rightarrow$ exclut HOL, PVS, …

On explique des langages simples (comme FUN) en termes d'un langage plus complexe (OCaml, Haskell, Agda, Coq).

Point d’étape

Point d'étape

Les langages fonctionnels (syntaxe, sémantique, typage) se mécanisent très bien dans l'ensemble …

… modulo quelques difficultés pour traiter les variables liées et l'alpha-conversion (équivalence à renommage près des variables liées).

Beaucoup de systèmes de types ont été mécanisés, avec des traits avancés comme

- Le polymorphisme de sous-typage (p.ex. $\texttt{bool} <: \texttt{int}$)
- Le polymorphisme paramétrique (p.ex. $\forall\alpha.\ \alpha \to \alpha$)
- Les types dépendants (p.ex. $\texttt{term}\ \Gamma\ \tau$)

Le prochain cours abordera les deux derniers sous l'angle de la théorie des types.

Bibliographie

Bibliographie

Deux livres de référence sur les langages fonctionnels typés :

- Benjamin Pierce. *Types and Programming Languages*. MIT Press, 2002.
- Robert Harper. *Practical Foundations for Programming Languages*. Cambridge University Press, 2016.

Mécanisations de langages fonctionnels typés :

- Approche extrinsèque, en Coq : Benjamin Pierce et al, *Software Foundations, volume 2 : Programming Languages Foundations*, `https://softwarefoundations.cis.upenn.edu/`.
- Approche intrinsèque, en Agda : Philip Wadler, *Programming Language Foundations in Agda*, `https://plfa.github.io/`