Université de Montréal

IFT-6172

Sémantiques des langages de programmation

Stefan Monnier (AA-2341)

monnier@iro.umontreal.ca

http://www.iro.umontreal.ca/~monnier/6172/

Définition d'un langage

Syntaxe

- Règles lexicales
- Grammaire

Sémantique

- Règles de typage (sémantique statique)
- Règles d'évaluation (sémantique dynamique)

$$\textit{List Char} \Rightarrow \textit{List Lexeme} \Rightarrow \textit{ASA} \Rightarrow \textit{AnnotatedASA} \Rightarrow \textit{Value}$$

Lambda-calcul

Inventé par Alonzo Church

La base de la théorie des langages de programmation

Inspiration de Lisp, Scheme, ML, Haskell, ...

$$e ::= c \mid x \mid \lambda x \to e \mid e_1\ e_2$$

Sémantique:

$$(\lambda x \to e_1)\ e_2 \quad \rightsquigarrow \quad e_1[e_2/x] \qquad \beta\text{-réduction}$$

Il y a aussi le renommage-α et la réduction-η

Chapitre 5 de Pierce

Types

Système de classification qui a deux origines indépendentes:

- Logique mathématique: introduits pour éviter des problèmes tels que le paradoxe de Russell
- Langages de programmation: besoin de distinguer des valeurs de natures différentes (e.g. différente taille)

Un *type* est comme un ensemble d'objets similaires

Similaires = acceptent les même opérations

Genres de typages

Un langage peut utiliser les types de deux manières:

- Typage dynamique: les *valeurs* portent leur type

  N'importe quelle variable peut contenir n'importe quelle valeur

- Typage statique: le type est associé aux *variables*

  Une variable ne peut contenir que des valeurs du type spécifié

Le typage est dit *fort* ou *faible* selon s'il est possible d'utiliser une opération sur une valeur du mauvais type

Certains langages ne sont simplement pas typés du tout!

Lambda-calcul typé simplement (STLC)

Aussi inventé par Alonzo Church (around 1940)

$$\begin{array}{lrcl} \textit{(types)} & \tau & ::= & \mathtt{Int} \mid \tau_1 \to \tau_2 \\ \textit{(expressions)} & e & ::= & c \mid x \mid \lambda x{:}\tau \to e \mid e_1\, e_2 \end{array}$$

Sémantique:

$$(\lambda x{:}\tau \to e_1)\, e_2 \rightsquigarrow e_1[e_2/x] \qquad \beta\textit{-réduction}$$

Il y a aussi le renommage-α et la réduction-η

Sémantique statique

Règles de typage — règles d'inférences

Jugements de la forme:

$$\Gamma \vdash e : \tau$$

- Γ Le (ou les) contexte(s)
- e L'expression (ou l'instruction) sur laquelle on porte un jugement
- τ Le type assigné à cette expression

Règles d'inférences de la forme

$$\frac{Hyp_1 \quad \ldots \quad Hyp_n}{Conclusion}$$

Sémantique statique de STLC

$$\Gamma, x : \tau \vdash x : \tau$$

$$\frac{\Gamma \vdash e_1 : \alpha \to \beta \qquad \Gamma \vdash e_2 : \alpha}{\Gamma \vdash e_1\ e_2 : \beta}$$

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x : \tau_1 \to e : \tau_1 \to \tau_2}$$

$$\Gamma \vdash n : \texttt{Int} \qquad\qquad \Gamma \vdash (+) : \texttt{Int} \to \texttt{Int} \to \texttt{Int}$$

Sémantique dynamique

Toutes sortes d'approaches; grandes catégories:

- Sémantique dénotationnelle

  Sorte de compilateur vers qqch comme le λ-calcul

- Sémantique opérationelle

  Sorte d'interpréteur. Peut être:

  – à *petits pas*

  $e \rightsquigarrow e'$

  – à *grands pas*

  $e \downarrow v$

- Sémantique axiomatique ...

Sémantique à petits pas pour STLC

Pour contrôler l'ordre d'évaluation, on défini la notion de *valeur*

$$\text{(\textit{values})} \quad v \;::=\; \lambda x{:}\tau \to e \mid c\, v_1 \ldots v_n$$

Réductions primitives:

$$(\lambda x{:}\tau \to e)\, v \rightsquigarrow e[v/x] \qquad \text{call-by-value } \beta\text{-reduction}$$

$$(+)\, n_1\, n_2 \rightsquigarrow n_3 \qquad \text{where } n_3 = n_1 + n_2$$

Règles de congruence (gauche à droite):

$$\frac{e_1 \rightsquigarrow e_1'}{e_1\, e_2 \rightsquigarrow e_1'\, e_2} \qquad \frac{e \rightsquigarrow e'}{v\, e \rightsquigarrow v\, e'}$$

Sémantique à petits pas avec trous

Règles de congruence remplacées par

- Définition syntaxique de “contexte” (*expression avec trou*)
- Une règle de congruence générique pour tous ces contextes

Par exemple:

(*expressions avec trou*) $E \quad ::= \quad \bullet \mid E\ e \mid v\ E$

$$\frac{e \rightsquigarrow e'}{E[e] \rightsquigarrow E[e']}$$

Où $E[e]$ signifie “E où $\bullet$ est remplacé par e”

Sémantique à grands pas pour STLC

$$c \downarrow c \qquad \lambda x : \tau \to e \downarrow \lambda x \to e$$

Cas de base

$$\frac{e_1 \downarrow \lambda x \to e \qquad e_2 \downarrow v' \qquad e[v/x] \downarrow v}{e_1\ e_2 \downarrow v}$$

Les primitives peuvent être introduites comme suit:

$$(+)\ n_1\ n_2 \downarrow n_3 \qquad (+)\ n \downarrow (+)\ n$$

$$\frac{e_1 \downarrow c\ v_1\ \ldots\ v_n \qquad e_2 \downarrow v_{n+1} \qquad c\ v_1\ \ldots\ v_n\ v_{n+1} \downarrow n}{e_1\ e_2 \downarrow v}$$

Polymorphisme en λ-calcul

Le λ-calcul typé avec polymorphisme s'appelle *System F*

Cœur des langages OCaml/Haskell (en théorie et en pratique)

$$e ::= c \mid x \mid \lambda x : \tau \to e \mid e_1\, e_2 \mid \Lambda t \to e \mid e[\tau]$$

$$\tau ::= \texttt{Int} \mid \tau_1 \to \tau_2 \mid t \mid \forall t.\tau$$

La fonction identité est en fait traduite comme suit:

$$\mathit{id} :: \forall \alpha.\alpha \to \alpha$$

$$\mathit{id} = \Lambda\alpha \to \lambda x \to x$$

$$\mathit{réponse} = \mathit{id}[\texttt{Int}]\, 42$$

Inventé en 1972/1974 par Girard/Reynolds

Sémantique statique de System F

$$\Gamma, x : \tau \vdash x : \tau$$

$$\frac{\Gamma \vdash e_1 : \alpha \to \beta \qquad \Gamma \vdash e_2 : \alpha}{\Gamma \vdash e_1\ e_2 : \beta}$$

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x : \tau_1 \to e : \tau_1 \to \tau_2}$$

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \Lambda t \to e : \forall t.\tau} \qquad \frac{\Gamma \vdash e : \forall t.\tau_1}{\Gamma \vdash e[\tau_2] : \tau 1[\tau_2/t]}$$

Curry-Howard

Intime connection entre logique et langages de programmation

$$\frac{\Gamma \vdash e_1 : \alpha \to \beta \qquad \Gamma \vdash e_2 : \alpha}{\Gamma \vdash e_1\ e_2 : \beta} \quad \text{vs} \quad \frac{P_1 \Rightarrow P_2 \qquad P_1}{P_2}$$

La règle de typage de l'application correspond au *modus ponens*

La β-réduction correspond au *cut-elimination*

Type = Proposition; Programme = Preuve

$$\frac{\Gamma \vdash e : \forall t.\tau_2}{\Gamma \vdash e[\tau_1] : \tau_2[\tau_1/t]} \quad \text{vs} \quad \frac{\forall x.P}{P[e/x]}$$

Il n'existe pas de fonction de type $\forall t_1, t_2.t_1 \to t_2$