Conception de langage
Projet 2006: note de synthèse

On va donner la sémantique d'un petit *langage source* que l'on décrit intuitivement ainsi :
- Déclarations : constantes, variables, fonctions (possiblement récursives) et procédures (possiblement récursives).
- Instructions : affectation, appel de procédure, bloc (avec déclarations locales, possiblement vide), boucle générale (Loop), break (pour sortir des boucles), alternative (If-Else) ;
- Expressions : constantes (lexicales), variables, appel de fonction, conditionnelle (IF), appel de fonction (incluant l'appel aux opérateurs lambda-calcul prédéfinis) ;
- un programme est une suite de déclarations (possiblement vide) suivie d'une instruction (en général un bloc).

Les fonctions sémantiques calculent un (gros) terme du lambda-calcul qui exprime le sens des programmes du langage source. On travail avec le lambda-calcul étendu par le point fixe, l'alternative, la structure de paires, avec les deux projections, et des types primitifs (entiers, chaînes de caratères, etc.) avec leurs opérateurs prédéfinis.

Domaines On définit un domaine Val dont les éléments sont des termes du lambda-calcul. On pose $Val = Data + Addr + Fun + Proc$, avec
- $Data$: ensemble des *données* calculables (booléens, entiers, chaînes, etc.) ;
- $Addr$: ensemble des *adresses mémoires* (par exemples, des entiers naturels) ;
- Fun : ensemble des lambda-termes qui représente les fonctions du langage source ;
- $Proc$: ensemble des lambda-termes qui représente les procédures du langage source ;

On rappelle que l'union disjointe (+) est réalisée en utilisant des couples dont le premier terme (par exemple un entier) sert à distinguer les divers types d'éléments ($Data$, $Addr$,etc.) et le second est l'élément lui-même.

Il faut définir les fonctions d'injections (constructeurs) et les fonctions de discrimination (reconnaisseurs) par des termes du lambda-calcul. Ici, il faut définir les injections inData inAddr inFun inProc, les fonctions de discrimination isData isAddr isFun isProc. Il faut égaement définir une fonction de projection (accès), ici, on définira valOf.

Données sémantiques Outre les arbres de syntaxe abstraite, les fonctions sémantiques manipulent trois types d'objets : les continuations ($Cont$), les environnements (Env) et les mémoires (Mem). Un environnement associe un nom à une valeur. Une mémoire associe une adresse à une donnée. Une continuation est une fonction des l'ensemble des mémoires dans l'ensemble des mémoires.

Il faut représenter les continuations, les environnement et les mémoires par de termes du lambda-calcul.
- continuation : on écrit le lambda-terme correspondant à la fonction voulue, par exemple, la continuation triviale est $\lambda m.m$ (on l'appelera κ_0) ;
- environnement : soit Str l'ensemble des chaînes de caractères, un couple de $Str \times Val$ représente une association en un nom et une valeur ; Env est l'ensemble des suites d'éléments de $Str \times Val$ (listes d'association) ;
- mémoire : soit Nat l'ensemble des entiers naturels, un couple de $Nat \times Data$ est une association entre une adresse et une donnée ; Mem est l'ensemble des suites d'éléments de $Nat \times Data$ (listes

d'association).

Les fonctions sémantiques manipulent les éléments de Env et Mem : ajout/modification de liaison, accès aux valeurs/données stockées. Il faut définir, comme des termes du lambda-calcul, les fonctions correspondantes. Ici, on définira eset(ajout à l'environnement) eget(accès à l'environnement) et mset(ajout/modification de la mémoire) mget(accès à la mémoire).

On utilisera des expressions de la forme ERROR : ... pour signaler les anomalies sémantiques. On peut par exemple utiliser des chaînes de caractères (les lambda-termes qui les représentent).

Fonctions sémantiques On rappelle et précise les fonctions sémantiques du cours.

$$
\begin{array}{ll}
\mathbf{P}: & \textsc{Prog} \to Env \to Mem \to Mem \\
\mathbf{D}: & \textsc{Dec} \to Env \to Mem \to Env \\
\mathbf{Ds}: & \textsc{Decs} \to Env \to Mem \to Env \\
\mathbf{S}: & \textsc{Stat} \to Cont \to Env \to Mem \to Mem \\
\mathbf{Ss}: & \textsc{Stat} \to Cont \to Env \to Mem \to Mem \\
\mathbf{E}: & \textsc{Exp} \to Env \to Mem \to Data
\end{array}
$$

– Programme :

$$\mathbf{P}[[Ds\ S]], \sigma, \mu \quad = \quad \mathbf{S}[[S]], \kappa_0, (\mathbf{Ds}[[Ds]], \sigma, \mu), \mu$$

– Déclarations :

$$
\begin{array}{lcl}
\mathbf{D}[[\texttt{Cst}\ x\ \texttt{:=}\ E\ \texttt{;}]], \sigma, \mu & = & (\mathsf{eset}\ x\ (\mathsf{inData}\ (\mathbf{E}[[E]], \sigma, \mu))\ \sigma) \\
\mathbf{D}[[\texttt{Var}\ x\ \texttt{;}]], \sigma, \mu & = & (\mathsf{eset}\ x\ (\mathsf{inAddr}\ newAddr)\ \sigma) \\
\mathbf{D}[[\texttt{Fun}\ (x_1 \ldots x_n)\ \texttt{=}\ E\ \texttt{;}]], \sigma, \mu & = & (\mathsf{eset}\ f\ (\mathsf{inFun}(\lambda m, v_1, \ldots, v_n.(\mathbf{E}[[E]], \sigma_f, m))) \\
 & & \text{avec } \sigma_f = (\mathsf{eset}\ x_1\ (\mathsf{inData}(v_1))\ \ldots (\mathsf{eset}\ x_n\ (\mathsf{inData}(v_n))\ \sigma)\ldots) \\
\mathbf{D}[[\texttt{FunRec}\ (x_1 \ldots x_n)\ \texttt{=}\ E\ \texttt{;}]], \sigma, \mu & = & (\mathsf{eset}\ f\ (\mathsf{inFun}(\lambda m.!f.\lambda v_1, \ldots, v_n.(\mathbf{E}[[E]], \sigma_f, m))) \\
 & & \text{avec } \sigma_f = (\mathsf{eset}\ x_1\ (\mathsf{inData}(v_1))\ \ldots (\mathsf{eset}\ x_n\ (\mathsf{inData}(v_n))\ \sigma)\ldots) \\
\mathbf{D}[[\texttt{Proc}\ p\texttt{(}x_1\texttt{,}\ldots\texttt{,}x_n\texttt{)}\ \texttt{=}\ S\ \texttt{;}]], \sigma, \mu & = & (\mathsf{eset}\ p\ (\mathsf{inProc}(\lambda m, v_1, \ldots, v_n.(\mathbf{S}[[S]], \kappa_0, \sigma_p, m))) \\
 & & \text{avec } \sigma_p = (\mathsf{eset}\ x_1\ (\mathsf{inData}(v_1))\ \ldots (\mathsf{eset}\ x_n\ (\mathsf{inData}(v_n))\ \sigma)\ldots) \\
\mathbf{D}[[\texttt{ProcRec}\ p\texttt{(}x_1\texttt{,}\ldots\texttt{,}x_n\texttt{)}\ \texttt{=}\ S\ \texttt{;}]], \sigma, \mu & = & (\mathsf{eset}\ p\ (\mathsf{inProc}(!p.\lambda m, v_1, \ldots, v_n.(\mathbf{S}[[S]], \kappa_0, \sigma_p, m))) \\
 & & \text{avec } \sigma_p = (\mathsf{eset}\ x_1\ (\mathsf{inData}(v_1))\ \ldots (\mathsf{eset}\ x_n\ (\mathsf{inData}(v_n))\ \sigma)\ldots)
\end{array}
$$

$$
\begin{array}{lcl}
\mathbf{Ds}[[]], \sigma, \mu & = & \sigma \qquad \text{(si } Ds \text{ est vide)} \\
\mathbf{Ds}[[D\ Ds]], \sigma, \mu & = & \mathbf{Ds}[[Ds]], (\mathbf{D}[[D]], \sigma, \mu), \mu
\end{array}
$$

– Instructions :

$$
\begin{array}{lcl}
\mathbf{S}[[x\ \texttt{:=}\ E\ \texttt{;}]],\kappa,\sigma,\mu & = & (\textsf{if}\ (\textsf{isAddr}\ a)\\
 & & \quad (\kappa\ (\textsf{mset}\ (\textsf{valOf}\ a)\ (\mathbf{E}[[E]],\sigma,\mu)\ \mu))\\
 & & \quad \textsf{ERROR: not a variable})\\
 & & \text{avec } a = (\textsf{eget}\ x\ \sigma)\\
\mathbf{S}[[p\texttt{(}E_1\texttt{,}\ \ldots\ \texttt{,}E_n\texttt{)}]],\kappa,\sigma,\mu & = & (\textsf{if}\ (\textsf{isProc}\ t)\\
 & & \quad (\kappa\ ((\textsf{valOf}\ t)\ \mu\ (\mathbf{E}[[E_1]],\sigma,\mu)\ldots(\mathbf{E}[[E_n]],\sigma,\mu)))\\
 & & \quad \textsf{ERROR: not a procedure})\\
 & & \text{avec } t = (\textsf{eget}\ x\ \sigma)\\
\mathbf{S}[[\texttt{Begin}\ Ds\ Ss\ \texttt{End}\ \texttt{;}]],\kappa,\sigma,\mu & = & \mathbf{Ss}[[Ss]],\kappa,(\mathbf{Ds}[[Ds]],\sigma,\mu)\mu\\
\mathbf{S}[[\texttt{Loop}\ S]],\kappa,\sigma,\mu & = & (\kappa\ (!k.\lambda m.(\mathbf{S}[[S]],k,\sigma,m)\ \mu))\\
\mathbf{S}[[\texttt{Break}\ \texttt{;}]],\kappa,\sigma,\mu & = & \mu\\
\mathbf{S}[[\texttt{If}\ E\ S_1\ \texttt{Else}\ S_2]],\kappa,\sigma,\mu & = & (\textsf{if}\ (\mathbf{E}[[E]],\sigma,\mu)\\
 & & \quad (\mathbf{S}[[S_1]],\kappa,\sigma,\mu)\\
 & & \quad (\mathbf{S}[[S_2]],\kappa,\sigma,\mu))\\
 & & \\
\mathbf{Ss}[[]],\kappa,\sigma,\mu & = & (\kappa\ \mu) \qquad (\text{si } Ds \text{ est vide})\\
\mathbf{Ss}[[S\ Ss]],\kappa,\sigma,\mu & = & \mathbf{S}[[S]],(\lambda m.(\mathbf{Ss}[[Ss]],\kappa,\sigma,m)),\sigma,\mu
\end{array}
$$

– Expressions :

$$
\begin{array}{lcl}
\mathbf{E}[[c]],\sigma,\mu & = & c \qquad (\text{pour toute constante lexicale – entier, chaîne, etc.})\\
\mathbf{E}[[x]],\sigma,\mu & = & (\textsf{if}\ (\textsf{isAddr}\ t)\\
 & & \quad (\textsf{mget}\ (\textsf{valOf}\ t)\ \mu)\\
 & & \quad (\textsf{valOf}\ t))\\
\mathbf{E}[[\texttt{(IF}\ E_1\ E_2\ E_3\texttt{)}]],\sigma,\mu & = & (\textsf{if}\ (\mathbf{E}[[E_1]],\sigma,\mu)\ (\mathbf{E}[[E_2]],\sigma,\mu)\ (\mathbf{E}[[E_3]],\sigma,\mu))\\
\mathbf{E}[[\texttt{(OP}\ E_1\ldots E_n\texttt{)}]],\sigma,\mu & = & (\textsf{OP}\ (\mathbf{E}[[E_1]],\sigma,\mu)\ldots(\mathbf{E}[[E_n]],\sigma,\mu)) \qquad (\text{pour tout opérateur OP prédéfini})\\
\mathbf{E}[[(f\ E_1\ldots E_n\texttt{)}]],\sigma,\mu & = & (\textsf{if}\ (\textsf{isFun}\ t)\\
 & & \quad ((\textsf{valOf}\ t)\ (\mathbf{E}[[E_1]],\sigma,\mu)\ldots(\mathbf{E}[[E_n]],\sigma,\mu))\\
 & & \quad \textsf{ERROR: not a function})\\
 & & \text{avec } t = (\textsf{eget}\ f\ \sigma)
\end{array}
$$

Mise en œuvre Je décris (notez le passage du «on» au «je») ci-dessous, à gros traits, la façon dont j'ai implanté la sémantique ci-dessus. Le langage d'implantation est Ocaml.

Avant d'entrer dans la description, un petit mot sur les *niveaux de langage* : la fonction d'évaluation du lambda-calcul utilise un environnement, les fonctions sémantiques manipulent un environnement ! Il faut bien distinguer les deux : l'environnement pour l'évaluation du lambda-calcul est une structure définie dans le langage d'implantation (ici, Ocaml), c'est un argument de la fonction d'évaluation du lambda-calcul ; l'environnement des fonctions sémantiques est un terme du lambda-calcul, ce sera un argument des fonctions sémantiques (comme les continuations et les mémoires).

J'ai défini un type `lterm` (module `Lb_type`) pour les termes du lambda-calcul étendu et une fonction d'évaluation `eval` (module `Lb_eval`).

J'ai défini un *environnement initial* pour pour la fonction `Lb_eval.eval` qui contient les définition des fonctions eset, eget, mset et mget ainsi que les définitions les injections, discrimination, projections inData, isData, etc. valOf. Je l'ai appelé `Lb_utils.lb_env0`.

J'ai défini, pour les fonctions sémantiques, une continuation, un environnement et une mémoire initiaux appelés respectiovement `Lb_utils.kap0`, `Lb_utils.sg0`, `Lb_utils.mu0`. La continuation initiale est l'identité, la mémoire initiale est vide, l'environnement initiale contient la définition des primitives pour les listes (`CONS CAR CDR NIL`).

Les arbres de syntaxe abstraites sont définies dans le module `Lsrc_ast`. Il y a un type pour chaque catégorie syntaxique. L'analyseur syntaxique produit un arbre syntaxique pour les programmes. Il est im-

planté par la fonction `Lsrc_parser.prog`.

J'ai défini pour chaque fonction sémantique la fonction correspondante. Elles sont définies dans le module `Lsrc_sem`. Voici la signature de cet ensemble de fonctions :

```
val semProg :
  Lsrc_ast.prog -> Lb_type.lterm -> Lb_type.lterm -> Lb_type.lterm
val semDec :
  Lsrc_ast.dec -> Lb_type.lterm -> Lb_type.lterm -> Lb_type.lterm
val semDecs :
  Lsrc_ast.dec list -> Lb_type.lterm -> Lb_type.lterm -> Lb_type.lterm
val semStat :
  Lsrc_ast.stat -> Lb_type.lterm -> Lb_type.lterm -> Lb_type.lterm -> Lb_type.lterm
val semStats :
  Lsrc_ast.stat list -> Lb_type.lterm -> Lb_type.lterm -> Lb_type.lterm -> Lb_type.lterm
val semExp :
  Lsrc_ast.exp -> Lb_type.lterm -> Lb_type.lterm -> Lb_type.lterm
```

Pour les adresses, j'ai été un peu brutal et j'ai défini deux fonctions de remise à zéro des adresses (`reset_addr`) et qui donne une nouvelle adresse (`new_addr`).

Enfin, j'ai défini la fonction principale de l'application qui prend en argument un de fichier contenant le code d'un programme du langage source et l'évalue. En voici le code (Ocaml) :

```
let eval_prog f =
 let ic = open_in f in
 let lexbuf = Lexing.from_channel ic in
 let ast = Lsrc_parser.prog Lsrc_lexer.token lexbuf in
 let lb = Lsrc_sem.semProg ast sg0 mu0 in
   reset_addr();
   Lb_eval.eval lb lb_env0 []
```