UPMC/master/info/4I503 APS
Notes de cours

P. Manoury

Janvier 2017

Statiquement, un *programme* est un fichier: code source ou exécutable. Un programme a également une *dynamique* qui est son comportement lors de son exécution. Le lien entre la donnée statique d'un programme et sa dynamique est l'objet de la *sémantique*. On peut anticiper certaines propriétés du comportement dynamique des programmes par *analyse* de la donnée statique des programmes.

Programme La donnée statique d'un programme correspond à son code source ou a son code compilé (exécutable ou code-octet). On traitera dans ce cours du code source des programmes.

Un code source est simplement une suite de caractères, en général stockée dans un ensemble de fichiers textes. Toutefois, toute suite de caractères n'est pas un code source. En effet, les codes sources des programmes doivent respecter les règles d'un langage. Ces règles définissent la la *syntaxe* des langages de programmation.

La définition de la syntaxe d'un langage comprend deux éléments: la définition d'un *lexique* et la définition d'une *grammaire*. Le lexique est l'ensemble des suites de caractère unitaires des langages, l'ensemble des *mots* et *symboles* utilisables dans le langage. Ces unités de langages sont des *lexèmes*. La grammaire énonce les règles d'agencements des lexémes. Elle définit l'ensemble des suites de lexèmes qui appartiennent au langage.

Lorsqu'elle sont formalisées, les définitions du lexique et de la grammaire d'un langage permettent la génération de fonctions d'analyse de suites de caractères et de suites de lexèmes capables de décider si une suite de caractères appartient ou non au langage défini. Ce genre d'analyse des suites de caractères est appelée *analyse syntaxique*. D'un point de vue théorique, on établit une relation entre les définitions formelles et des automates (automates à états finis, automates à pile, etc.) qui sont des objets facilement implémentables en machine.

Cette génération est opérationnalisée par des outils logiciels comme `lex` et `yacc`, pour ne parler que des ancêtres.

Sémantique Le comportement dynamique d'un programme est induit par la traitement de la donnée statique d'un programme par un micro-processeur, un interprète de code-octet, voire un interprète de code source.

En pratique, les programmes sont réalisés dans l'*intention* d'obtenir un certain comportement. Celui-ci est manifesté par la production d'un *résultat* ou d'un *effet* pour le dispositif sur lequel le programme est exécuté: affichage, production ou modification de fichier, etc. Dans tous les cas, on peut dire que le but d'un programme est de produire la modification d'un *état mémoire*; la mémoire de vive l'ordinateur, ou la mémoire de masse d'un système de fichiers.

La sémantique a pour objet d'exprimer la relation entre les constructions syntaxiques d'un programme et la production de résultats ou d'effets. Cette relation est définie en liant la construction syntaxique des langages de programmation avec leur effet sur le *contexte d'exécution* des programmes. Ce contexte est constitué par une certaine organisation en mémoire des *valeurs* manipulées par les programmes. La définition

de la sémantique est *dirigée par la syntaxe*, en ce sens que, *grosso modo*, à chaque règle de contruction syntaxique des langages est associée une règle d'exécution, ou règle *évaluation*.

Donner la sémantantique d'un langage de programmation, c'est donner la spécification formelle d'un interpréteur de code source.

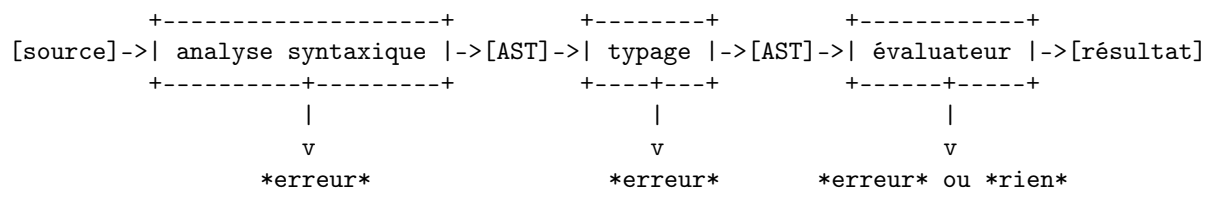
Analyse L'analyse opère sur la donnée statique des programmes, souvent son code source. Elle a pour but d'anticiper leur comportement dynamique. Ceci permet en particulier de savoir diagnostiquer par avance qu'un programme est suceptible de provoquer une erreur d'exécution.

Le plus répandu de ce type d'analyse est réalisé par les mécanismes de *vérification de type* qui accompagnent les compilateurs des langages de programmation. Les programmes manipulent et produisent des valeurs. Ces valeurs peuvent être classifiées par *types*. Manipuler ou produire une valeur n'appartenant pas au type attendu par une opération du programme peut provoquer une erreur d'exécution ou une incohérence du contexte d'exécution. La vérification de type, ou *typage*, appliqué à un programme peut éliminer ce risque d'erreur ou d'incohérence.

Le principe définition de l'analyse de type est similaire à celui mis en œuvre pour la sémantiue: l'analyse est guidée par la syntaxe. *Grosso modo*, à chaque règle syntaxique des langages est associée une règle de vérification de cohérence de type des valeurs manipulées et produites.

Mise en œuvre On mettra en œuvre les trois éléments de traitement des programmes que sont l'analyse syntaxique, l'analyse de type, et l'évaluation par la réalisation des composants logiciels correspondant.

Schéma de principe



L'analyse syntaxique produit une `*erreur*` lorsque la suite de caractères donnée en entrée n'appartient pas au langage. Lorsque ce n'est pas le cas, le processus d'analyse syntaxique produira uen *arbre de syntaxte abstrait* du langage (*Abstract Syntax Tree*). Celui-ci sera donnée sous la forme d'un *terme prolog* (une chaîne de caractères) ou sous la forme d'une structure en mémoire (structure arborescente, instance de classe).

L'analyse de typage sera réalisée en PROLOG, dans ce cas, votre analyse syntaxique devra produire un terme PROLOG.

Le langage d'implantation de l'évaluateur est laissé libre. Vous adapterez la sortie de votre analyse syntaxique en conéquence.

1 Le langage APS0

On appelle *APS0* notre premier langage. Il contient le noyau impératif minimal que l'on rencontre dans la plupart des langages de programmations. Ses programmes ne manipulent que des valeurs entières ou booléennes. Les composants du langages sont des opérations de base sur les entiers et les booléens; les instrcutions d'affectation, d'alternative et de boucle non bornée; des déclarations de variables et de constantes; des blocs encapsulant déclarations et instructions.

1.1 Syntaxe

La syntaxe est définie par un *lexique* et une *grammaire*. Le lexique définit les *unités lexicales* du langage ou *lexèmes*. On y trouve les *symboles réservés* et les *mots clef* du langage (que l'on peut aussi appeler *mots réservés*; ainsi que les ensemble de lexèmes utilisés pour désigner les *constantes numériques* et les identificateurs.

Lorsque l'on définit la grammaire du langage, les unités lexicales sont considérées comme des *symboles terminaux* dans le formalisme de la définition des grammaires. Les règles de grammaires définissent des ensemble de suites d'unités lexicales comme la valeur de *symboles non terminaux*.

Pour présenter la syntaxe, lexique et grammaire, on utilise un certain nombre de *convention typographiques*. Les *symboles réservés* et les *mots clef* du langage sont indiqués en caractère `machine à écrire`. Les ensembles de lexèmes, comme les *constantes numériques* ou les *identificateurs* sont indiqués en caractères sans serif. Les *symboles non terminaux* de la définition de la grammaire sont indiqués en caractères PETITES CAPITALES.

Lexique Dans la définition du lexique, on trouve l'énumération d'un certain nombres de suites de caractères explicitement données (symboles réservés et mots clef) ou la définition d'un ensemble des suites de caractères en utilisant les opérations sur les suites de caractères caractérisant les *expressions rationnelles* (*regular expressions*). Ici, nous emprunterons le formalisme de `lex` pour décrire les expressions rationnelles, plus précisément, sa déclinaison dans l'outils `ocamllex`.

Symboles réservés `[ ] ( ) ;`

Mots clef `VAR CONST SET IF WHILE  bool int  true false not and or  eq lt add sub mul div`

Constantes numériques num défini par *('-'?)['0'-'9']+*

Identificateurs ident défini par *(['a'-'z''A'-'Z'])(['a'-'z''A'-'Z''0'-'9'])**

Pour être opérationnelle, c'est-à-dire, permettre de reconnaître et d'isoler les lexèmes dans un flux de caractères, la définition du lexique spécifie également quels sont les *caractères séparateurs*. Il s'agit, en général de l'espace, la tabulation, le passage à la ligne et le retour chariot.

Gammaire La grammaire définit le sous ensemble des suites de lexèmes que l'on veut retenir pour le langage.

La définition, non encore formalisée des programmes appartenant au langage *APS0* est la suivante:

UN PROGRAMME est une suite de commandes enclose entre crochets.

UNE SUITE DE COMMANDES est

- soit une instruction;
- soit une instruction suivie d'un point virgule, suivi d'une suite de commandes;
- soit une déclaration suivie d'un point virgule, suivi d'une suite de commandes.

UNE DÉCLARATION est

- soit une déclaration de constante constituée du mot clef `CONST` suivi d'un identificateur, suivi d'un type, suivi d'une expression;
- soit une déclaration de variable constituée du mot clef `VAR` suivi d'un identificateur, suivi d'un type.

Un type est

- soit le mot réservé `int`;
- soit le mot réservé `bool`.

Une instruction est

- soit une affectation constituée du mot clef `SET` suivi d'un identificateur, suivi d'une expression;
- soit une alternative constituée du mot clef `IF` suivi d'une expression, suivi de deux programmes;
- soit une boucle constituée du mot clef `WHILE` suivi d'une expression, suivi d'un programme.

Une expression est

- soit le mot réservé `true`;
- soit le mot réservé `false`;
- soit une constante numérique;
- soit un identificateur (nom de constante ou de variable);
- soit l'application constituée du mot réservé `not` suivi d'une expression, le tout entre parenthèses;
- soit l'application constituée du mot réservé `and` suivi de deux expressions, le tout entre parenthèses;
- soit l'application constituée du mot réservé `or` suivi de deux expressions, le tout entre parenthèses;
- soit l'application constituée du mot réservé `eq` suivi de deux expressions, le tout entre parenthèses;
- soit l'application constituée du mot réservé `lt` suivi de deux expressions, le tout entre parenthèses;
- soit l'application constituée du mot réservé `add` suivi de deux expressions, le tout entre parenthèses;
- soit l'application constituée du mot réservé `sub` suivi de deux expressions, le tout entre parenthèses;
- soit l'application constituée du mot réservé `mul` suivi de deux expressions, le tout entre parenthèses;
- soit l'application constituée du mot réservé `div` suivi de deux expressions, le tout entre parenthèses.

La formalisation adoptée pour la définition de la grammaire est une *BNF*, pour *Backus-Naur Form*. La grammaire définit un ensemble de symboles *non terminaux* dont chacun correspond à une catégorie d'éléments de langage de programmation: les déclarations, les instructions, les expressions, etc. Les symboles *terminaux* correspondent aux unités lexicales. Un non terminal est distingué comme *entrée* de la syntaxe. C'est lui qui définit l'ensemble des programmes du langage. Ici, c'est Prog.

Les définitions des non terminaux sont *mutuellement récursives*.

```
Prog  ::=  [ Cmds ]
Cmds  ::=  Stat | Dec ; Cmds | Stat ; Cmds
Dec   ::=  VAR ident Type
       |   CONST ident Type Expr
Stat  ::=  SET ident Expr
       |   IF Expr Prog Prog
       |   WHILE Expr Prog
Expr  ::=  true | false
       |   num | ident
       |   ( not Expr ) | ( and Expr Expr ) | ( or Expr Expr )
       |   ( eq Expr Expr ) | ( lt Expr Expr )
       |   ( add Expr Expr ) | ( sub Expr Expr )
       |   ( mul Expr Expr ) | ( div Expr Expr )
Type  ::=  bool | int
```

1.2 Typage

Les programmes écrits en *APS0* manipulent des valeurs entières ou booléennes au moyen des opérateurs de base fournis par le langage. Ces opérateurs sont donnés pour réaliser les fonctions arithmétiques et logiques usuelles: addtion, soustraction, multiplication, division, négation, disjonction, conjonction. La sémantique donnera ce sens aux symboles du langage que l'on souhaite associer à ces opérations. En tant que fonctions arithmétiques et logiques, elles possèdent un *domaine* de définition qui spécifie à quel ensemble de valeurs doivent appartenir ses arguments et un *codomaine* qui spécifie à quel ensemble appartient le résultat de leur application. Dans le langage, ces ensembles de valeurs correspondent aux *types* `bool` et `int`. L'analyse de type d'une expression consiste à déterminer si l'application des opérateurs est conforme à la spécification du domaine des fonctions correspondantes. Si tel est le cas, l'analyse de type permet également de déterminer le type du résultat de l'application.

Ceci vaut pour les expressions du langage. Pour ce qui est des instructions, on introduit un type qui leur est spécifique: le type `void`. Dans *APS0*, les instructions n'ont pas à proprement parler de valeurs: elles ne valent que par l'effet qu'elles ont sur la mémoire. Cependant, elles utilisent des expressions et doivent obéirent à certaine contraintes:

- dans une affectation, le type de la variable affectée et celui de l'expression qui lui donnera sa valeur doivent être cohérents;
- dans une alternative ou une boucle, l'expression qui conditionne leur comportement doit être une expression booléenne.

Du point de vue de l'analyse de type, on considère les instructions comme des «fonctions» dont on doit déterminer le type des «arguments». Si ceux-ci sont corrects, on assigne à l'instruction le type `void`.

Les identificateurs utilisés comme constantes symboliques ou symboles de variables n'appartiennent *a priori* à aucun type en propre mais les déclarations sont là pour leur en assigner un. L'ensemble des assignations de types à ces identificateurs est appelé *contexte de typage*.

Contexte de typage Un contexte de typage est donc une association entre identificateurs et types. Abstraitement, un contexte de typage est une fonction de l'ensemble des identificateurs, dans l'ensemble des types. Dans cette perspective, si G est un contexte, alors $G(x)$ est le type associé à x par G. Si t est le type associé à x par G, on a $G(x) = t$. Si $G(x)$ n'est pas défini, on pose $G(x) = \mathit{undef}$.

On note $G; (x : t)$ l'extension de G avec la liaison $(x : t)$. On a

$$\begin{array}{lcl} G; (x : t)(x) & = & t \\ G; (x : t)(y) & = & G(y) \quad \mathsf{si}\ y \neq x \end{array}$$

Jugement de typage L'objectif de l'analyse de type est d'émettre un *jugement de typage*: *« dans le contexte G, l'expression e est de type t »*. Ce jugement est une relation entre un contexte G, une expression e et un type t. On note $G \vdash e : t$. Pour typer tous les éléments d'un programme, il faut également une relation entre un contexte G, une instruction s et le type void. On note la relation $G \vdash s : \mathsf{void}$. Enfin, il faut une relation entre un contexte G, une suite de commandes cs et le type void. On note $G \vdash cs : \mathsf{void}$.

Les relations de typages sont définies de manière *mutuellement récursives*. Les clauses de définition de la relation de typage sont appelées *règles de typage*.

Règles de typage

Les règles de typages sont définies selon la catégorie syntaxique des éléments des lanages de programmation: expressions, instructions, suites de commandes (déclarations et instructions). On considère les trois types bool, int et void. On nomme chaque règle.

Dans ce qui suit: G désigne un contexte de typage, e, e_1, e_2 désignent des expressions; s une instruction; cs, une suite de commandes; blk, blk_1 et blk_2 désignent des blocs de commande (entrée PROG de la grammaire); n désigne une constante numérique (un élément de num); x, désigne un identificateur (un élément de ident). Les symboles réservées, les mots clef et les mots réservés sont cités en police `machine à écrire`.

Constantes

(TRUE) $G \vdash$ `true` : bool

(FALSE) $G \vdash$ `false` : bool

(NUM) $G \vdash n$: int

Booléens

(NOT) si $G \vdash e$: bool alors $G \vdash$ (`not` e) : bool

(AND) si $G \vdash e_1$: bool et $G \vdash e_2$: bool alors $G \vdash$ (`and` e_1 e_2) : bool

(OR) si $G \vdash e_1$: bool et $G \vdash e_2$: bool alors $G \vdash$ (`or` e_1 e_2) : bool

Comparaison

(EQ) si $G \vdash e_1$: int et $G \vdash e_2$: int alors $G \vdash$ (`eq` e_1 e_2) : bool

(LT) si $G \vdash e_1$: int et $G \vdash e_2$: int alors $G \vdash$ (`lt` e_1 e_2) : bool

Nombres

(ADD) si $G \vdash e_1$: int et $G \vdash e_2$: int alors $G \vdash$ (`add` e_1 e_2) : int

(SUB) si $G \vdash e_1$: int et $G \vdash e_2$: int alors $G \vdash$ (`sub` e_1 e_2) : int

(MUL) si $G \vdash e_1$: int et $G \vdash e_2$: int alors $G \vdash$ (`mul` e_1 e_2) : int

(DIV) si $G \vdash e_1$: int et $G \vdash e_2$: int alors $G \vdash$ (`div` e_1 e_2) : int

Variables

(ID) si $G(x) = t$ alors $G \vdash x : t$

Instructions

(SET) si $G \vdash x : t$ et $G \vdash e : t$, alors $G \vdash$ (`SET` x e) : void

(IF) si $G \vdash e$: bool et $G \vdash blk_1$: void et $G \vdash blk_2$: void alors $G \vdash$ (`IF` e blk_1 blk_2) : void

(WHILE) si $G \vdash e$: bool et $G \vdash blk$: void alors $G \vdash$ (`WHILE` e blk) : void

Suite de commandes

(STAT) si $G \vdash s$: void et $G \vdash cs$: void alors $G \vdash s$`;`cs : void

(CONST) si d est une déclaration de la forme (`CONST` x t e), si $G \vdash e : t$ et $G; (x : t) \vdash cs$: void alors $G \vdash d$`;`cs : void

(VAR) si d une déclaration de la forme (`VAR` x t), si $G; (x : t) \vdash cs$: void alors $G \vdash d$`;`cs : void

(END) $G \vdash []$: void, où [] est une suite vide.

Bloc (ou programme)

(PROG) si $G \vdash cs : \mathsf{void}$ alors $G \vdash [cs] : \mathsf{void}$

1.3 Sémantique opérationnelle

La première sémantique que nous donnons à notre langage est appelée *sémantique opérationnelle*. Plus précisément, nous présentons une variété de la sémantique opérationnelle appelée *sémantique naturelle* ou *sémantique à grand pas*.

La sémantique rend compte des aspects *fonctionnels* du langage: les expressions donnent des valeurs; et des aspects *impératifs* du langage: les instructions modifient un *état mémoire* et les déclarations étendent un *environnement*. La mémoire et l'environnement constituent le *contexte d'évaluation* des programmes.

Environnement À l'instar des contextes de typage, un environnement est une association entre identificateurs et valeurs ou adresses mémoires. Les constantes sont associées à leur valeur, les variables sont associées à leur adresse. Abstraitement, un environnement est une fonction dont le domaine est l'ensemble des identificateurs (ident) et, le codomaine est l'union de l'ensemble des valeurs manipulées par le langage et les adresses mémoire. Les valeurs sont soit les valeurs booléennes (nous les noterons $\#t$ et $\#f$) soit des valeurs numériques entières (nous noterons $\#n$ la avleur entière dénitée par la constante numérique n). L'ensemble des adresses est, pour l'instant, laissé totalement abstrait.

On notera $\#v$ pour désigner une valeur booléenne ou numérique, $@a$ pour désigner une adresse et w pour désigner une valeur ou une adresse. On note $[]$ l'environnement vide (fonction jamais définie).

Pour désigner la valeur ou l'adresse associée à l'identificateur x dans l'environnement r, on note simplement $r(x)$. Notez que $r(x)$ n'est pas toujours défini, auquel cas, on pose $r(x) = \mathit{error}$, en particulier: $[](x) = \mathit{error}$.

L'extension d'un environnement r consiste à y ajouter ou à redéfinir une association entre un identificateur x et une valeur ou adresse w. On note $r[x = w]$ et on pose:

$$\begin{array}{lcll} r[x = w](x) & = & w & \\ r[y = w](x) & = & r(y) & \text{si } y \neq x \end{array}$$

Mémoire On se donne une vision idéalisée de la mémoire comme une fonction qui associe des adresses à des valeurs.

Ainsi, on note $m(a)$ la valeur «contenue» à l'adresse a. La mémoire est une fonction partielle. Intuitivement, les adresses pour lesquelles la valeur de m n'est pas définie sont des adresses qui n'ont pas été allouées. On note, dans ce cas, $m(a) = \mathit{error}$, en particulier, $[](a) = \mathit{error}$, où $[]$ désigne une mémoire vide, la mémoire dans laquelle aucune adresse n'a été allouée.

Sur la mémoire, on définit deux opérations: l'extension et la modification.

L'allocation mémoire est modélisée par l'ajout d'une adresse à son domaine. On note $m[a = \mathit{new}]$.

La modification du «contenu» d'une adresse mémoire est notée $m[a := v]$. On pose:

$$\begin{array}{lcll} m[a = \mathit{new}][a := v] & = & m[a = v] & \\ m[a = v][a := v'] & = & m[a = v'] & \\ m[a = w][a' := v] & = & m[a' := v][a = w] & \text{si } a \neq a' \\ [][a := v] & = & \mathit{error} & \end{array}$$

Notez que l'opération de modification peut n'être pas définie si l'adresse a n'est pas dans le domaine de la mémoire.

On définit enfin une opération de *restriction* du domaine d'une mémoire m en fonction d'un environnement r. Le but de cette opération est de ne concerver dans le domaine de m que les identificateurs de r correspondant à des variables allouées. C'est une opération de désallocation. On note m/r. On pose

$$\begin{array}{lcll} (m/r)(a) & = & m(a) & \text{si il existe } x \text{ tel que } r(x) = @a \\ & = & \mathit{error} & \text{sinon} \end{array}$$

On remarque que si r est l'enrironnment vide [] alors $(m/[]) = m$.

Jugements sémantiques

Définir la sémantique des programmes, c'est définir les relations entre les contextes d'évaluation (environnement et mémoire), les éléments syntaxiques des programmes et des valeurs ou de nouveaux contextes d'évaluation. La définition de ces relations est dirigée par la syntaxe, leurs composantes dépendent des éléments de langage considérés.

expression relation entre un environnement, un état mémoire, une expression et une valeur. On note $r, m \vdash e \rightsquigarrow v$ pour *dans l'environnement r et la mémoire m, l'expression e donne la valeur v.*

instruction relation entre un environnement, un état mémoire, une instruction et un état mémoire. On note $r, m \vdash s \rightsquigarrow m'$ pour *dans l'environnement r et la mémoire m, l'instruction s produit la mémoire m'.*

déclaration relation entre un environnement, un état mémoire, une déclaration et un nouvel état mémoire et un nouvel environnement. On note $r, m \vdash d \rightsquigarrow r', m'$ pour: *dans l'environnement r et la mémoire m, la déclaration d produit l'environnement r' et la mémoire m'.*

suite de commandes relation entre un environnement, un état mémoire, une suite de commandes et un nouvel état mémoire. On note $r, m \vdash cs \rightsquigarrow m'$ pour *dans l'environnement r et la mémoire m, la suite de commandes cs produit la mémoire m'.*

bloc relation entre un environnement, un état mémoire, un bloc (de commandes), et un nouvel état mémoire. On note $m, r \vdash blk \rightsquigarrow m'$.

Sémantique des expressions

Constantes

(TRUE) $r, m \vdash \texttt{true} \rightsquigarrow \#t$

(FALSE) $r, m \vdash \texttt{false} \rightsquigarrow \#f$

(NUM) $r, m \vdash n \rightsquigarrow \#n$

Identificateurs Pour obtenir la valeur d'un identificateur, on accède à la valeur que lui donne l'environnement. S'il s'agit d'une adresse, on accéde à la valeur en mémoire. Les symboles # et @ servent à distinguer les valeurs immédiates (entiers et booléens) des adresses (variables).

(IMD) si $r(x) = \#v$ alors $r, m \vdash x \rightsquigarrow v$

(ADR) si $r(x) = @a$ alors $r, m \vdash x \rightsquigarrow m(a)$

Opérations booléennes On définit *in extenso* les opérations booléennes.

(NOT1) si $r, m \vdash e \rightsquigarrow \#t$ alors $r, m \vdash (\texttt{not}\ e) \rightsquigarrow \#f$

(NOT2) si $r, m \vdash e \rightsquigarrow \#f$ alors $r, m \vdash (\texttt{not}\ e) \rightsquigarrow \#t$

(OR1) si $r, m \vdash e_1 \rightsquigarrow \#t$ alors $r, m \vdash (\texttt{or}\ e_1\ e_2) \rightsquigarrow \#t$

(OR2) si $r, m \vdash e_1 \rightsquigarrow \#f$ et $r, m \vdash e_2 \rightsquigarrow v$ alors $r, m \vdash (\texttt{or}\ e_1\ e_2) \rightsquigarrow v$

(AND1) si $r, m \vdash e_1 \rightsquigarrow \#f$ alors $r, m \vdash (\texttt{and}\ e_1\ e_2) \rightsquigarrow \#f$

(AND2) si $r, m \vdash e_1 \rightsquigarrow \#t$ et $r, m \vdash e_2 \rightsquigarrow v$ alors $r, m \vdash (\texttt{and}\ e_1\ e_2) \rightsquigarrow v$

Les règles OR1 OR2 AND1 AND2 définissent une disjonction et une conjonction *séquentielles*. Les règles OR1 et AND1 spécifient que leur premier argument est évalué en premier et que, s'il vaut $\#t$ pour la disjonction ou $\#f$ pour la conjonction, la valeur de l'application est donnée sans qu'il soit besoin d'évaluer le second argument. C'est de cette manière que sont implémentés ces deux connecteurs booléens dans la plupart des langages de programmation.

Comparaisons On ne peut définir ici *in extenso* les opérations de comparaison. Mais l'on connaît, sur le domaine des entiers du moins, les relations d'égalité et d'infériorité. les règles sémantiques des opérateurs de comparaison consistent donc simplement à associer aux symboles d'opérateurs de comparaison `eq` et `lt` les relations arithmétiques d'égalité et d'infériorité.

(EQ1) si $r, m \vdash e_1 \rightsquigarrow v_1$ et $r, m \vdash e_2 \rightsquigarrow v_2$ et si $v_1 = v_2$ alors $r, m \vdash (\texttt{eq}\ e_1\ e_2) \rightsquigarrow \#t$

(EQ2) si $r, m \vdash e_1 \rightsquigarrow v_1$ et $r, m \vdash e_2 \rightsquigarrow v_2$ et si $v_1 \neq v_2$ alors $r, m \vdash (\texttt{eq}\ e_1\ e_2) \rightsquigarrow \#f$

(LT1) si $r, m \vdash e_1 \rightsquigarrow v_1$ et $r, m \vdash e_2 \rightsquigarrow v_2$ et si $v_1 < v_2$ alors $r, m \vdash (\texttt{lt}\ e_1\ e_2) \rightsquigarrow \#t$

(LT2) si $r, m \vdash e_1 \rightsquigarrow v_1$ et $r, m \vdash e_2 \rightsquigarrow v_2$ et si $v_2 \leq v_1$ alors $r, m \vdash (\texttt{lt}\ e_1\ e_2) \rightsquigarrow \#f$

Opérations arithmétiques Comme pour les comparaisons, on définit simplement les opérations arithmétiques par des ... opérations arithmétiques.

(ADD) si $r, m \vdash e_1 \rightsquigarrow v_1$ et $r, m \vdash e_2 \rightsquigarrow v_2$ et si $v = v_1 + v_2$ alors $r, m \vdash (\texttt{add}\ e_1\ e_2) \rightsquigarrow v$

(SUB) si $r, m \vdash e_1 \rightsquigarrow v_1$ et $r, m \vdash e_2 \rightsquigarrow v_2$ et si $v = v_1 - v_2$ alors $r, m \vdash (\texttt{sub}\ e_1\ e_2) \rightsquigarrow v$

(MUL) si $r, m \vdash e_1 \rightsquigarrow v_1$ et $r, m \vdash e_2 \rightsquigarrow v_2$ et si $v = v_1 \times v_2$ alors $r, m \vdash (\texttt{mul}\ e_1\ e_2) \rightsquigarrow v$

(DIV) si $r, m \vdash e_1 \rightsquigarrow v_1$ et $r, m \vdash e_2 \rightsquigarrow v_2$ et si $v = v_1 \div v_2$ alors $r, m \vdash (\texttt{div}\ e_1\ e_2) \rightsquigarrow v$

Sémantique des instructions

Affectation On vérifie dynamiquement que le membre gauche est modifiable.

(SET) si $r(x) = @a$ et si $r, m \vdash e \rightsquigarrow v$ alors $r, m \vdash (\texttt{SET}\ x\ e) \rightsquigarrow m[a := v]$

Alternative On a une règle pour chaque cas de condition. Notez que dans chaque cas, un seul des termes de l'alternative est évalué.

(ALT1) si $r, m \vdash e \rightsquigarrow \#t$ et $r, m \vdash blk_1 \rightsquigarrow m'$ alors $r, m \vdash (\texttt{IF}\ e\ blk_1\ blk_2) \rightsquigarrow m'$

(ALT2) si $r, m \vdash e \rightsquigarrow \#f$ et $r, m \vdash blk_2 \rightsquigarrow m'$ alors $r, m \vdash (\texttt{IF}\ e\ blk_1\ blk_2) \rightsquigarrow m'$

Boucle Ici également, il y a deux règles selon que la condition de boucle est vérifiée ou non.

(LOOP1) si $r, m \vdash e \rightsquigarrow \#t$ et $r, m \vdash blk \rightsquigarrow m'$ et $m' \vdash (\texttt{WHILE}\ e\ blk) \rightsquigarrow m''$ alors $r, m \vdash (\texttt{WHILE}\ e\ blk) \rightsquigarrow m''$

(LOOP0) si $r, m \vdash e \rightsquigarrow \#f$ alors $r, m \vdash (\texttt{WHILE}\ e\ blk) \rightsquigarrow m$

Sémantique des déclarations

On distingue dans l'environnement les valeurs immédiates (constantes) des adresses (variables).

(CONST) si $r, m \vdash e \rightsquigarrow v$ alors $r, m \vdash (\texttt{CONST}\ x\ t\ e) \rightsquigarrow (r[x = \#v], m)$

(VAR) $r, m \vdash (\texttt{VAR}\ x\ t) \rightsquigarrow (r[x = @a], m[a = new])$, avec $a \notin \mathrm{dom}(m)$.

Sémantique des suites de commandes

Il faut considérer la suite de commandes vide, que l'on note ε et que toute suite non vide de commandes est de la forme $c_1\,;\ldots;\varepsilon$. Pour toute suite de commande cs:

(NOP) $r, m \vdash \varepsilon \rightsquigarrow m$

(DEC) pour toute déclaration d, si $r, m \vdash d \rightsquigarrow (r', m')$ et si $r', m' \vdash cs \rightsquigarrow m''$ alors $r, m \vdash (d\,;cs) \rightsquigarrow m''$

(STAT) pour tout instruction s, si $r, m \vdash s \rightsquigarrow m'$ et si $r,' \vdash cs \rightsquigarrow m''$ alors $r, m \vdash (s\,;cs) \rightsquigarrow m''$

Sémantique des blocs

On «désalloue» les variables locales du bloc: les adresses allouées lors de l'évaluation du bloc sont supprimées du domaine de la mémoire résultant de l'évaluation du bloc; elles pourrons être «recyclées».

(BLOC) si $r, m \vdash (cs\,;\varepsilon) \rightsquigarrow m'$ alors $r, m \vdash \texttt{[}cs\texttt{]} \rightsquigarrow (m'/r)$

Programme

L'évaluation d'un programme est simplement l'évaluation de la suite de commandes qu'il contient avec un environnement initial et une mémoire initiale vides.

2 Sémantique dénotationnelle

La sémantique opérationnelle définit un ensemble de relations qui relient les contexte d'évaluation (environnement et mémoire), les constructions syntaxiques du langage et soit des valeurs, soit une mémoire, soit un contexte d'évaluation, selon la nature de l'unité syntatxique considérées.

La sémantique dénotationnelle reprend ces éléments, mais les relient en termes de *fonctions*: pour chaque catégorie syntaxique, on définit une *fonction sémantique*. Pour exprimer celà, on a besoin de quelques notions ensemblistes de base ainsi que d'une notation pour les fonctions.

2.1 Théorie algébrique naïve pour les ensembles

On ne cherche pas à *fonder* le concept d'ensemble. On cherche simplement à exprimer les relations et opérations utiles sur les ensembles. On admet donc que l'on connaôt déjà quelques ensembles, en particulier les booléens $\mathbb{B}$, dont les éléments seront notés tt et ff, et l'ensemble $\mathbb{N}$ des entiers naturels dont les élements seront notés comme d'habitude.

Appartenance On note $x \in X$ l'appartenance de *l'élément* x à *l'ensemble* X. On note $x \notin X$ la négation de $x \in X$.

Ensemble vide On se donne l'ensemble $\emptyset$ tel que quelque soit x, $x \notin \emptyset$. En d'autres termes, $\emptyset$ est l'enesemble vide.

Singleton Si x est un élement, on note $\{x\}$ l'ensemble contenant x, et uniquement lui. C'est-à-dire que quelque soit z, $z \in \{x\}$ si et seulement si $z = x$.

Inclusion On dit que l'ensemble X *est inclus* dans l'ensemble Y si et seulement si, quelque soit z, si $z \in X$ alors $z \in Y$. On note cette relation $X \subseteq Y$. On dit alors que X est un *sous-ensemble* ou encore que X est une *partie* de Y.

Paires et produit cartésien On note (x, y) la *paire ordonnée* formée par x et y. On se donnent les projections fst et snd telles que $\mathsf{fst}(x, y) = x$ et $\mathsf{snd}(x, y) = y$.

On note $X \times Y$ l'ensemble des paires (x, y) avec $x \in X$ et $y \in Y$. C'est-à-dire que, quelque soit z, $z \in X \times Y$ si et seulement si il existe $x \in X$ et $y \in Y$ tels que $z = (x, y)$.

Fonctions Du point de vue ensembliste, *une fonction de domaine X et de codomaine Y* est une partie f de $X \times Y$ telle que pour tout $(x, y) \in f$ et $(x', y') \in f$, $x \neq x'$; ou encore, pour tout $x \in X$, pour tout $y \in Y$ et $y' \in Y$, si $(x, y) \in f$ et $(x, y') \in f$ alors $y = y'$. C'est-à-dire que f associe aux éléments de X un unique élément de Y. On note $X \to Y$ l'ensemble des fonctions de domaine X et de codomaine Y. Selon l'usage, on note $f : X \to y$ au lieu de $f \in X \to Y$.

On dit que $f : X \to Y$ est une *fonction totale* lorsque, pout tout $x \in X$, il existe $y \in Y$ tel que $(x, y) \in f$. Si tel n'est pas le cas, c'est-à-dire si il existe $x \in X$ tel que, quelque soit $y \in Y$, $(x, y) \notin f$, alors, on dit que f est *partielle*.

La notation usuelle $f(x) = y$ est une variante notationnelle pour $(x, y) \in f$.

ATTENTION: la notation $f(x)$ ne désigne rien si, quelque soit $y \in Y$, $f(x) \neq y$, c'est-à-dire, $(x, y) \notin f$.

Union On note $X \cup Y$ l'ensemble formé par les éléments de X et les éléments de Y. C'est à dire que, quelque soit z, $z \in X \cup Y$ si et seulement si $z \in X$ ou $z \in Y$.

Union disjointe On note $X \oplus Y$ l'ensemble $(\{0\} \times X) \cup (\{1\} \times Y)$. C'est l'ensemble formé des paires de la forme $(0, x)$ avec $x \in X$ ou $(1, y)$ avec $y \in Y$. C'est-à-dire que, quelque soit z, $z \in X \oplus Y$ si et seulement si, il existe $x \in X$ tel que $z = (0, x)$ ou il existe $y \in Y$ tel que $z = (1, y)$.

On définit $inX : X \to X \oplus Y$ et $inY : Y \to X \oplus Y$ tels que pour tout $x \in X$, $inX(x) = (0, x)$ et pour tout $y \in Y$, $inY(y) = (1, Y)$. Pour tout $x \in X$ et $y \in Y$, on a $inX(x) \in X \oplus Y$ et $inY(y) \in X \oplus Y$. On peut voir l'ensemble $X \oplus Y$ comme l'ensemble des éléments de la forme $inX(x)$ ou $inY(y)$ pour tout $x \in X$ et $y \in Y$.

On définit $outX : X \oplus Y \to Y$ et $outY : X \oplus Y \to Y$ par $outX(0, x) = x$ et $outY(1, y) = y$. Notez que ni $outX(1, z)$ ni $outY(0, z)$ ne sont définis. On a que que pour tout $x \in X$, $outX(inX(x)) = x$ et, pour tout $y \in Y$, $outY(inY(y)) = y$.

On définit enfin, les fonctions booléennes $isX : X \oplus Y \to \mathbb{B}$ et $isY : X \oplus Y \to \mathbb{B}$ par $isX(0, x) = \mathsf{tt}$ et $isY(1, y) = \mathsf{tt}$.

2.2 Un langage de fonctions

On définit un langage d'expressions fonctionnelles basé sur la notation du λ-calcul.

Le langage de fonction du lambda calcul est formé d'une ensemble de symboles de constantes et de symboles de variable ainsi que des symboles réservés (,) et λ (par commodité on utilisera également le point .). Si x désigne un symbole de variable et c un symbole de constante, la grammaire de ce langage est la suivante:

```
T  ::=  x
     |  c
     |  ( T T )
     |  λ x . T
     |  ( T )
```

On appelle *termes* ou *λ-termes* les expressions du langage du λ-calcul. Un terme de la forme $(t_1\ t_2)$ est appelé une *application*. Un terme de la forme $\lambda x.t$ est appelé une *abstraction*. L'abstraction $\lambda x.t$ représente la fonction qui à x associe t. Par exemple, avec les symboles de constantes + et 1, on aura $\lambda x.(+\ x\ 1)$ pour le fonction successeur sur les entiers.

Un terme de la forme $(\lambda x.t\ u)$ est appelé un *redex*: c'est l'application de l'abstraction (fonction) $\lambda x.t$ au terme (argument) u. La valeur de l'application $(\lambda x.t\ u)$ est égale à la valeur du terme obtenu en remplaçant

x par u dans t. On note ce terme $t[u/x]$[1]. On a $(\lambda x.t\ u) = t[u/x]$.

On ajoute à l'ensemble des termes ceux formés à l'aide d'un combinateur pour l'alternative (if) ainsi qu'un *combinateur de point fixe* (fix):

$$\begin{array}{rcl} \mathrm{T} & ::= & \ldots \\ & | & (\mathsf{if}\ t\ t\ t) \\ & | & !x.t \end{array}$$

On pose:

- $(\mathsf{if}\ \mathsf{tt}\ t_1\ t_2) = t_1$
- $(\mathsf{if}\ \mathsf{ff}\ t_1\ t_2) = t_2$
- $!x.t = t[!x.t/x]$

On se donne un ensemble d'abréviations de termes:

- on écrit $(t_1\ t_2\ t_3)$ au lieu de $((t_1\ t_2)\ t_3)$, $(t_1\ t_2\ t_3\ t_4)$ pout $(((t_1\ t_2)\ t_3)\ t_4)$, etc.
  ATTENTION: $(t_1\ t_2\ t_3)$ n'st pas une abréviation pour $(t_1\ (t_2\ t_3))$.
- on écrit $\lambda x_1.x_2.t$ pour $\lambda x_1.\lambda x_2.t$, $\lambda x_1.x_2.x_3.t$ pour $\lambda x_1.\lambda x_2.\lambda x_3.t$,etc.

On définit également un ensemble de *macros notations*
Redex

$$\text{let } x = u \text{ in } t \quad \equiv \quad (\lambda x.t\ u)$$

$$\text{let } (x,y) = t \text{ in } u \quad \equiv \quad \begin{array}{l} \text{let } x = (\mathsf{fst}\ t) \text{ in} \\ \text{let } y = (\mathsf{snd}\ t) \text{ in} \\ \quad u \end{array}$$

Unions disjointes (types sommes) et macros pour le *filtrage*. Soit $t \in X \oplus Y$, on pose

$$\text{let } inX(x) = t \text{ in } u \quad \equiv \quad \begin{array}{l} (\mathsf{if}\ isX(t) \\ \quad \text{let } x = outX(t) \text{ in } u \\ \quad \bot) \end{array}$$

$$\begin{array}{l} \text{case } t \text{ of} \\ \quad inX(x)\text{: } t_1 \\ |\ \ inY(y)\text{: } t_2 \end{array} \quad \equiv \quad \begin{array}{l} (\mathsf{if}\ isX(t) \\ \quad \text{let } x = outX(t) \text{ in } t_1 \\ \quad (\mathsf{if}\ isY(t) \\ \quad\quad \text{let } y = outY(t) \text{ in } t_2 \\ \quad\quad \bot)) \end{array}$$

2.3 Domaines sémantiques

- N ensemble de valeurs numériques ($\mathbb{N}$)
- B ensemble des valeurs booléennes ($\mathbb{B}$)
- A ensemble d'adresses
- $V = B \cup N$ valeurs immédiates, numériques ou booléennes
- $E = \mathsf{ident} \to (A \oplus V)$ environnements
- $S = A \to V$ la mémoire

[1] Il faut donner une définition précise de cette notation, mais nous nous contenterons pour l'instant de cette «définition» intuitive

Fonctions utiles sur les domaines On suppose données deux fonctions $\beta : \{\texttt{true}, \texttt{false}\} \to I\!B$ et $\nu : \mathsf{num} \to I\!N$ qui projettent les unités lexicales dans les domaines de valeur $I\!B$ et $I\!N$.

On se donne une fonction d'allocation $alloc : S \to (A \times S)$ telle que $alloc(\sigma) = a, \sigma'$ si et seulement si $a \notin \mathrm{dom}(\sigma)$ et $\mathrm{dom}(\sigma') = \mathrm{dom}(\sigma) \cup \{a\}$. La valeur $\sigma'(a)$ est indéterminée.

On reprend pour les environnements ρ et la mémoire σ les opérations $\rho[x = w]$, avec $w \in V \oplus A$, $\sigma[a := v]$ avec $a, \in A$ et $v \in V$, aisni que (σ/ρ).

2.4 Équations sémantiques

Expressions

$\mathbf{E} : \textsc{Expr} \to E \to S \to V$

soit $b \in \{\texttt{true}, \texttt{false}\}$, $n \in \mathsf{num}$, $x \in \mathsf{ident}$;
soit $\rho \in E$ et $\sigma \in S$

$$\mathbf{E}[[b]]\rho\sigma = \beta(b)$$

$$\mathbf{E}[[n]]\rho\sigma = \nu(n)$$

$$\mathbf{E}[[x]]\rho\sigma = \text{case } \rho(x) \text{ of } inA(a) : \sigma(a) \mid inV(v) : v$$

$$\mathbf{E}[[(\texttt{not } e)]]\rho\sigma = (\mathsf{if}\ (\mathbf{E}[[e]]\rho\sigma)\ \mathsf{ff}\ \mathsf{tt})$$

$$\mathbf{E}[[(\texttt{and } e_1\ e_2)]]\rho\sigma = (\mathsf{if}\ (\mathbf{E}[[e_1]]\rho\sigma)\ (\mathbf{E}[[e_2]]\rho\sigma)\ \mathsf{ff})$$

$$\mathbf{E}[[(\texttt{or } e_1\ e_2)]]\rho\sigma = (\mathsf{if}\ (\mathbf{E}[[e_1]]\rho\sigma)\ \mathsf{tt}\ (\mathbf{E}[[e_2]]\rho\sigma))$$

$$\mathbf{E}[[(\texttt{eq } e_1\ e_2)]]\rho\sigma = (\mathbf{E}[[e_1]]\rho\sigma) =_V (\mathbf{E}[[e_2]]\rho\sigma)$$

$$\mathbf{E}[[(\texttt{lt } e_1\ e_2)]]\rho\sigma = (\mathbf{E}[[e_1]]\rho\sigma) <_V (\mathbf{E}[[e_2]]\rho\sigma_)$$

$$\mathbf{E}[[(\texttt{add } e_1\ e_2)]]\rho\sigma = (\mathbf{E}[[e_1]]\rho\sigma) +_V (\mathbf{E}[[e_2]]\rho\sigma)$$

$$\mathbf{E}[[(\texttt{sub } e_1\ e_2)]]\rho\sigma = (\mathbf{E}[[e_1]]\rho\sigma) -_V (\mathbf{E}[[e_2]]\rho\sigma)$$

$$\mathbf{E}[[(\texttt{mul } e_1\ e_2)]]\rho\sigma = (\mathbf{E}[[e_1]]\rho\sigma) \times_V (\mathbf{E}[[e_2]]\rho\sigma)$$

$$\mathbf{E}[[(\texttt{div } e_1\ e_2)]]\rho\sigma = (\mathbf{E}[[e_1]]\rho\sigma) \div_V (\mathbf{E}[[e_2]]\rho\sigma)$$

Intructions

$\mathbf{S} : \textsc{Stat} \to E \to S \to S$

$$\mathbf{S}[[\texttt{SET } x\ e]]\rho\sigma = \text{let } inA(a) = \rho(x) \text{ in } \sigma[a := \mathbf{E}[[e]]\rho\sigma]$$

$$\mathbf{S}[[\texttt{IF } e\ \texttt{[}cs_1\texttt{]}\ \texttt{[}cs_2\texttt{]}]]\rho\sigma = (\mathsf{if}\ (\mathbf{E}[[e]]\rho\sigma)\ (\mathbf{B}[[cs_1]]\rho\sigma)\ (\mathbf{B}[[cs_2]]\rho\sigma))$$

$$\mathbf{S}[[\texttt{WHILE } e\ \texttt{[}cs\texttt{]}]]\rho\sigma = (!w.\lambda s.(\mathsf{if}\ (\mathbf{E}[[e]]\rho s)\ (w\ (\mathbf{B}[[cs]]\rho s))\ s)\ \sigma)$$

Déclarations

$\mathbf{D} : \textsc{Dec} \to E \to S \to (E \times S)$

$$\begin{array}{lcl} \mathbf{D}[[\texttt{CONST}\ x\ t\ e]]\rho\sigma & = & (\rho[x = inV(\mathbf{E}[[e]]\rho\sigma))],\ \sigma) \\ \\ \mathbf{D}[[\texttt{VAR}\ x\ t]]\rho\sigma & = & \text{let } a, \sigma' = alloc(\sigma) \text{ in} \\ & & \quad (\rho[x = inA(a)],\ \sigma') \end{array}$$

Suites de commandes

$\mathbf{Cs} : \textsc{Cmds} \to E \to S \to S$

soit $d \in \textsc{Dec}$, $s \in \textsc{Stat}$ et $cs \in \textsc{Cmds}$

$$\begin{array}{lcl} \mathbf{Cs}[[d\,;cs]]\rho\sigma & = & \text{let } \rho', \sigma' = \mathbf{D}[[d]] \text{ in} \\ & & \quad \mathbf{Cs}[[cs]]\rho'\sigma' \\ \\ \mathbf{Cs}[[s\,;cs]]\rho\sigma & = & \text{let } \sigma' = \mathbf{S}[[s]]\rho\sigma \text{ in} \\ & & \quad \mathbf{Cs}[[cs]]\rho, \sigma' \end{array}$$

Blocs

$\mathbf{B} : \textsc{Prog} \to E \to S \to S$

soit $cs \in \textsc{Cmds}$

$$\begin{array}{lcl} \mathbf{B}[[\texttt{[}cs\texttt{]}]]\rho\sigma & = & \text{let } \sigma' = \mathbf{Cs}[[cs]]\rho\sigma \text{ in} \\ & & \quad (\sigma'/\rho) \end{array}$$