SU/FS/master/info/MU4IN503 APS
Notes de cours

P. Manoury

Janvier 2020

Contents

1 APS0: noyau fonctionnel

Le langage *APS0* est essentiellement un *langage d'expressions*. On y manipule des valeurs entières et booléennes. Ces valeurs sont désignées par des symboles de constantes ou peuvent être obtenues par *application* d'opérateurs primitifs (les opérations arithmétiques et booléennes usuelles, les opérateurs de comparaison).

Le langage *APS0* est *pleinement fonctionnel* en ce sens que les expressions y désignent non seulement les valeurs arithmétiques ou booléennes, mais également des *fonctions*. Dans le langage *APS0*, une fonction est une valeur comme une autre: elle peut être le paramètre d'une fonction, voire, le résultat d'un calcul.

Le langage *APS0* permet de *définir* des constantes ou des fonctions; c'est-à-dire, permet de *lier* à des *noms* (ou *identificateurs*) des valeurs, fonctionnelles ou non.

Légère exception à son caractère fonctionnel, le langage *APS0* contient une, et une seule, *instruction* qui a pour but de produire l'affichage d'une valeur (arithmétique).

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 ensembles 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.

1.1.1 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 `CONST FUN REC ECHO bool int true false not and or eq lt add sub mul div if`

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

Identificateurs ident défini par *(['a'-'z''A'-'Z'])(['a'-'z''A'-'Z''0'-'9'])**
dont on exclut les mots clef.

Appelons

- cbool l'ensemble des mots clef : `true false`
- oprim l'ensemble de mots clef: `not and or eq lt add sub mul div`
- tprim l'ensemble de mots clefs `bool int`

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.

1.1.2 Grammaire

La grammaire définit le sous ensemble des suites de lexèmes que l'on veut retenir pour le langage; c'est-à-dire, définit l'ensemble des suites de lexèmes acceptables comme un *programme*.

Appelons PROG cet ensemble. On pose qu'il est constitué de l'ensemble des *suites de commandes* encloses entre les symboles `[` et `]`. Appelons CMDS l'ensemble des suites de commandes. On définit alors PROG de la manière suivante:

PROG ::= `[` CMDS `]`

Une *commande* est soit une *déclaration*, soit une *instruction*. Nous appelons DEC l'ensemble des déclarations et STAT l'ensemble des instructions. Les *suites de commandes* du langages *APS0* sont des suites de déclarations et d'instructions séparées par le symbole `;` (point virgule). On fixe que le dernier membre d'une suite de commandes du langage *APS0* est toujours une instruction et qu'il n'y a pas de suite de commandes vide dans ce langage. On définit alors l'ensemble CMDS de la manière suivante:

CMDS ::= STAT
| DEC `;` CMDS
| STAT `;` CMDS

C'est une définition récursive: par exemple, une suite de commande est une déclaration suivi d'un point virgule, lui-même suivi d'une suite de commandes. Cette définition récursive est *bien fondée* car il existe un *cas de base* à la définition: la suite consistant en une seule instruction. Naturellement, cette définition sera effectivement bien fondée lorsque l'on aura défini STAT.

Une *déclaration* est soit la déclaration, en fait, la *définition* d'une constante, soit la définition d'une fonction, possiblement récursive. Les suites de lexèmes correspondant aux déclarations sont introduites par les mots clefs `CONST`, `FUN` et `REC` selon les cas possibles. Une déclaration de constante doit comprendre la mention du nom de la constante (un *identificateur*, élément de ident), le type de la constante et sa valeur qui est donnée sous forme d'une *expression*. Une fonction est définie à l'aide des mêmes éléments auxquels ont ajoute la liste des paramètres formels de la fonction, avec leur type.

Un *type* est soit le symbole d'un type de base (élément de tprim), soit un type fonctionnel qui indique la suite des types des arguments de la fonction et le type du résultat de l'application de la fonction. La suite de type des arguments et le type du résultat sont séparés par le symbole `->`; les types des arguments sont séparés par le symbole `*`. Appelons TYPE l'ensemble des types pour *APS0* et TYPES l'ensemble des suites de types pour les arguments. On définit de manière *mutuellement récursives* ces deux ensembles de la manières suivante:

TYPE ::= tprim
| `(` TYPES `->` TYPE `)`

TYPES ::= TYPE
| TYPE `*` TYPES

Cette double définition est *bien fondée* car (intuitivement) la 1ère clause de la définition l'ensemble TYPE est un cas de base (tprim), donc la définition de TYPE est bien fondée; et alors la première clause de la définition de TYPES en constitue également un cas de base.

Dans une déclaration de fonction, un *argument* indique le nom de l'argument (identificateur, élément de ident) ainsi que son type (élément de TYPE). Appelons ARG l'ensemble de telles paires. On le définit par:

ARG ::= ident `:` TYPE

Dans une suite d'arguments, les paires indiquant le nom et le type des arguments sont séparés par le symbole `,` (virgule). Appelons ARGS l'ensemble de ces suites. Il est définit par:

ARGS ::= ARG
| ARG , ARGS

Appelons EXPR l'ensemble des *expressions* du langage *APS0*. Nous pouvons alors définir l'ensemble DEC de la manière suivante:

DEC ::= `CONST` ident TYPE EXPR
| `FUN` ident TYPE `[` ARGS `]` EXPR
| `FUN REC` ident TYPE `[` ARGS `]` EXPR

L'ensemble STAT des instruction de *APS0* se définit en une seule clause:

STAT ::= `ECHO` EXPR

Pour achever la définition de l'ensemble des programmes de *APS0*, reste à définir l'ensemble des expressions EXPR. Une expression du langage *APS0* est

- soit un symbole de constante booléenne (`true` ou `false`), soit un symbole de constante numérique (élément de num), soit un identificateur (élément de ident);
- soit *l'application* du mot clef `if` à trois expressions;
- soit *l'application* d'un symbole d'opération primitives (élément de oprim) à une *suite d'expressions*;
- soit, de manière générale, *l'application* d'une expression à une suite d'expressions;
- soit *l'abstraction* fonctionnelle d'une expression obtenue en préfixant l'expression par une suite de déclaration d'arguments (élément de ARGS).

L'application est notée selon le mode *préfixé complètement parenthésé*. Les définitions de l'ensemble des expressions (EXPR) et de l'ensemble des suites d'expressions, que nous appelons EXPRS, sont données ainsi:

EXPR ::= cbool | num | ident
| `(` `if` EXPR EXPR EXPR `)`
| `(` oprim EXPRS `)`
| `[` ARGS `]` EXPR
| `(` EXPR EXPRS `)`
EXPRS ::= EXPR
| EXPR EXPRS

Ce sont deux *définitions mutuellement récursives* dont la bonne fonction repose sur les cas de base de la définition de EXPR qui sont cbool, num et ident.

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 et celui-ci doit correspondre au codomaine de la fonction associée à l'opérateur. Cela suppose, bien entendu que nous connaissions le type des opérateurs de base, ce qui est le cas. Nous verrons tout-à-l'heure comment.

En revanche, les identificateurs utilisés comme constantes symboliques, noms de fonctions ou noms de paramètres n'appartiennent *a priori* à aucun type en propre. Toutefois, leur déclaration, soit comme constante (*cf* DEC), fonction (*cf* (FUN) ou encore comme paramètre formel (*cf* ARG) sont là pour leur en assigner un. L'ensemble des *assignations de types* de ces identificateurs seront données par un *contexte de typage*.

La vérification de type appliquée aux expressions permet de sélectionner un sous-ensemble des éléments de EXPR qui *ont un sens* dans la perspective de leur évaluation. Ainsi, l'addition d'une valeur numérique à une valeur booléenne n'a pas, dans notre conception de langage *APS0*, de sens; de même l'application de l'opérateur d'addition à trois expressions, fussent-elle des expressions arithmétiques. Ainsi, l'analyse de type que nous envisageons va-t-elle rejeter les expressions `(add 0 true)` et `(add 1 2 3)`, qui sont *syntaxiquement correctes* mais que nous ne soumettrons pas au processus d'évaluation.

Dans le même ordre d'idée, nous vérifierons que l'expression suivant l'instruction d'affichage `ECHO` est bien une expression arithmétique, c'est-à-dire, de type `int`.

Pour ce qui est des déclarations, l'analyse de type est chargée de vérifier la cohérence de la déclaration. Par exemple, on vérifiera que le type annoncé d'un symbole de constante est bien celui de l'expression qui lui est associée dans la déclaration. Si cette cohérence est vérifiée, le contexte de typage est enrichi avec l'assignation du type déclaré au symbole défini.

Contexte de typage Un contexte de typage est une association entre symboless (opérateurs, contantes booléennes ou identificateurs) et types. Posons $\mathsf{sym} = \mathsf{cbool} \cup \mathsf{oprim} \cup \mathsf{ident}$. Formellement, on modélise l'association réalisée dans un contexte de typage par une *fonction partielle* de l'ensemble des symboles de sym, dans l'ensemble des types TYPE. Posons $G = \mathsf{sym} \to \text{TYPE}$. Dans cette perspective, si Γ est un contexte, alors $\Gamma(x)$ désigne le type associé à x par Γ, lorsqu'il est défini.

Lors de l'analyse de type d'un programme, le contexte de typage évolue: chaque déclaration introduit une nouvelle association entre identificateur et type.

On note $\Gamma[x : t]$ l'extension de Γ avec la liaison de x à t. L'extension $\Gamma[x : t]$ est la fonction de G telle que:

- $\Gamma[x : t](x) = t$
- $\Gamma[x : t](y) = \Gamma(y)$ lorsque x et y sont des symboles différents.

Abréviation: on écrit $\Gamma[x_1 : t_1; x_2 : t_2]$ comme abréviation de $\Gamma[x_1 : t_1][x_2 : t_2]$. Plus généralement, on écrit $\Gamma[x_1 : t_1; \ldots; x_n : t_n]$ pour $\Gamma[x_1 : t_1] \ldots [x_n : t_n]$.

On définit un contexte particulier que l'on note Γ_0. Il donne le type des opérateurs primitifs et des deux constantes booléennes. On pose que:

$$
\begin{array}{lcl}
\Gamma_0(\texttt{true}) & = & \texttt{bool} \\
\Gamma_0(\texttt{false}) & = & \texttt{bool} \\
\Gamma_0(\texttt{not}) & = & \texttt{bool -> bool} \\
\Gamma_0(\texttt{and}) & = & \texttt{bool * bool -> bool} \\
\Gamma_0(\texttt{or}) & = & \texttt{bool * bool -> bool} \\
\Gamma_0(\texttt{eq}) & = & \texttt{int * int -> bool} \\
\Gamma_0(\texttt{lt}) & = & \texttt{int * int -> bool} \\
\Gamma_0(\texttt{add}) & = & \texttt{int * int -> int} \\
\Gamma_0(\texttt{sub}) & = & \texttt{int * int -> int} \\
\Gamma_0(\texttt{mul}) & = & \texttt{int * int -> int} \\
\Gamma_0(\texttt{div}) & = & \texttt{int * int -> int}
\end{array}
$$

et que $\Gamma_0(x)$ n'est pas défini pour tout x de ident.

C'est ainsi que l'on «connaîtra» le type des opérateurs primitifs et des constantes booléennes. Cette manière de faire nous évitera d'avoir à poser une règle de typage pour chaque application d'un opérateur primitif.

Jugement de typage L'objectif de l'analyse de type est d'émettre un *jugement de typage*, par exemple: *« dans le contexte* Γ*, l'expression* e *est de type* t *»*. Formellement, ce jugement est une *relation* entre un contexte Γ, une expression e et un type t. L'analyse de type d'un programme complet utilise plusieurs catégories de jugements de typage. Ces catégories suivent *grosso modo* les catégories syntaxiques du langage *APS0*: les expressions, les instructions, les déclarations et les suites de commandes.

L'analyse de type des expressions de *APS0* doit garantir la correction de l'utilisation des composants de l'expression vis-à-vis des types. Lorsque cette correction est vérifiée, on peut assigner un type à l'expression. Un jugement de typage pour les expressions de *APS0* est l'énoncé de cette assignation. Les jugments acceptables sont définis formellement par la relation notée $\vdash_{\text{Expr}}$ qui est un sous ensemble des triplets de $G \times \text{Expr} \times \text{Type}$. On écrit les jugements de typages, pour les expressions, sous la forme: $\Gamma \vdash_{\text{Expr}} e : t$.

Dans *APS0*, le type associé à l'instruction `ECHO` n'est pas un élément de Type. L'instruction ne produit pas de valeur, elle a simplement un *effet*. Nous lui assignons donc un type spécifique que nous notons *void*. La relation de typage des instruction est notée $\vdash_{\text{Stat}}$, c'est un sous ensemble de $G \times \text{Stat} \times \{\textit{void}\}$. On écrit: $\Gamma \vdash_{\text{Stat}} s : \textit{void}$

Nous traitons les déclarations pour ce qu'elles sont: des déclarations. Dans cette perspective, l'analyse de type d'une déclaration n'associe pas un type à une déclaration, mais un nouveau contexte de typage dans lequel est introduit l'association entre l'identificateur déclaré et son type à condition naturellement que la cohérence entre le type déclaré et le type de l'expression utilisée dans la déclaration a été établie. La relation de typage pour les déclaration est notée $\vdash_{\text{Dec}}$, c'est un sous ensemble de $G \times \text{Dec} \times G$. On écrit: $\Gamma \vdash_{\text{Dec}} d : \Gamma'$.

Dans *APS0*, les suites de commandes qui ne contiennnet que l'instruction `ECHO` ou des déclarations ne produisent pas de valeur. Nous leur assignerons le type *void*. La relation de typage pour les suites de commandes est notée $\vdash_{\text{Cmds}}$, c'est un sous ensemble de $G \times \text{Cmds} \times \{\textit{void}\}$. On écrit: $\Gamma \vdash_{\text{Cmds}} cs : \textit{void}$

Un programme de *APS0* étant simplement une suite de commandes, on lui assignera également le type *void*. On note $\vdash$ la relation de typage, c'est un sous ensemble de $\text{Prog} \times \{\textit{void}\}$. On écrit: $\vdash p : \textit{void}$.

En résumé, pour définir la *discipline de type* de *APS0*, nous devons définir les cinq relations suivantes:

1. $\vdash$ dans $\text{Prog} \times \{\textit{void}\}$, on écrit $\vdash p : \textit{void}$.
2. $\vdash_{\text{Cmds}}$ dans $G \times \text{Cmds} \times \{\textit{void}\}$, on écrit: $\Gamma \vdash_{\text{Cmds}} cs : \textit{void}$
3. $\vdash_{\text{Dec}}$ dans $G \times \text{Dec} \times G$, on écrit: $\Gamma \vdash_{\text{Dec}} d : \Gamma'$.
4. $\vdash_{\text{Stat}}$ dans $G \times \text{Stat} \times \{\textit{void}\}$, on écrit: $\Gamma \vdash_{\text{Stat}} s : \textit{void}$
5. $\vdash_{\text{Expr}}$ dans $G \times \text{Expr} \times \text{Type}$, on écrit: $\Gamma \vdash_{\text{Expr}} e : t$.

Les définitions de ces relations énoncent les conditions à satisfaire pour que chaque relation soit satisfaite. Chaque relation correspond à une construction syntaxique du langage et, pour chaque cas, les conditions sont données en fonction des sous cas de construction syntaxique, des déclarations ou des expressions, par exemple. L'analyse de type visant à établir la satisfaction des relations de typage est donc *guidée par la syntaxe*.

Un programme p est dit *bien typé* s'il satisfait la relation $\vdash$, c'est-à-dire que l'on a pu vérifier que $\vdash p : \textit{void}$.

1.2.1 Expressions

On définit la relation $\vdash_{\text{Expr}}$ selon les *cas de construction* des expressions. Les cas de construction des expressions sont donnés par les clauses des règles syntaxiques qui définissent les expressions. Nous «factoriserons» toutefois les deux règles d'application de la grammaire (`(` oprim Exprs `)` et `(` Expr Exprs `)`) puisque le traitement des constructions syntaxiques associées est similaire.

Les constantes numériques sont décrétées de type `int`:

(NUM) si $n \in$ num alors $\Gamma \vdash_{\text{Expr}} n$: `int`

Un symbole a le type que lui donne le contexte. Si le contexte ne définit aucun type pour le symbole considéré, le typage échoue.

(SYM) si $x \in$ sym et si $\Gamma(x) = t$ alors $\Gamma \vdash_{\text{Expr}} x : t$

Une abstraction fonctionnelle `[`x_1 : $t_1, \ldots, x_n$: t_n`]`e est correctement typée lorsque l'expression e est correctement typée dans un contexte où les arguments ont le type indiqué dans l'abstraction. Dans ce cas, on peut assigner son type à l'abstraction:

(ABS) si $\Gamma[x_1 : t_1; \ldots; x_n : t_n] \vdash_{\text{Expr}} e : t$ alors $\Gamma \vdash_{\text{Expr}}$ `[`$x_1 : t_1, \ldots, x_n : t_n$`]`$e : t_1$ `*` $\ldots$ `*` t_n `->` t

Une application est correctement typée si le type de l'expression en position de fonction dans l'application a un type cohérent avec celui des expressions en position d'arguments de l'application. Dans ce cas, on peut donner le type de l'application:

(APP) si $\Gamma \vdash_{\text{Expr}} e : t_1$ `*` $\ldots$ `*` t_n `->` t, si $\Gamma \vdash_{\text{Expr}} e_1 : t_1$, $\ldots$ et si $\Gamma \vdash_{\text{Expr}} e_n : t_n$
alors $\Gamma \vdash_{\text{Expr}}$ `(`$e\ e_1 \ldots e_n$`)` : t

Enfin, l'expression alternative (`if`) est correctement typée si son premier argument est de type `bool` et l'ont peut assigner un même type aux deux alternants. Le type de l'expression alternative est le type commun des alternants.

(IF) si $\Gamma \vdash_{\text{Expr}} e_1$: `bool`, si $\Gamma \vdash_{\text{Expr}} e_2 : t$ et si $\Gamma \vdash_{\text{Expr}} e_3 : t$
alors $\Gamma \vdash_{\text{Expr}}$ `(if` $e_1\ e_2\ e_3$`)` : t

L'opérateur d'alternative est *polymorphe*: il a potentiellement une infinité de types. Toutefois, tout ceux-ci doivent respecter la forme `(bool *` t `*` t `->` t`)` avec $t \in$ Type.

1.2.2 Instruction

On vérifie simplement que l'expression dont on veut afficher la valeur est de type `int`. Dans ce cas, on assigne le type *void* à l'instruction d'affichage:

(ECHO) si $\Gamma \vdash_{\text{Expr}} e$: `int` alors $\Gamma \vdash_{\text{Stat}}$ (`ECHO` e) : *void*

1.2.3 Déclarations

On énonce une règle pour chaque cas de déclaration: constante, fonction, fonction récursive.

Une déclaration de constante est bien typée lorsque l'on peut assigner à l'expression qui définit la constante, le type déclaré pour cette constante. Dans ce cas, on ajoute une nouvelle liaison au contexte de typage:

(CONST) si $\Gamma \vdash_{\text{Expr}} e : t$ alors $\Gamma \vdash_{\text{Dec}}$ (`CONST` $x\ t\ e$) : $\Gamma[x : t]$

Une déclaration de fonction est correctement typée lorsque le contexte enrichi des assignations de type déclarées pour les arguments de la fonction permettent d'assigner le type déclaré du résultat de la fonction à l'expression qui définit la fonction. On construit le type de la fonction en combinant (avec `*` et `->`) les types déclarés des paramètres et le type attendu du résultat de l'application:

(FUN) si $\Gamma[x_1 : t_1; \ldots; x_n : t_n] \vdash_{\text{Expr}} e : t$
alors $\Gamma \vdash_{\text{Dec}}$ (`FUN` x t `[`x_1`:`t_1`,` $\ldots$ `,`x_n`:`t_n`]` e) : $\Gamma[x : t_1$ `*` $\ldots$ `*` t_n `->` $t]$

Une déclaration de fonction récursive est correctement typée dans les mêmes conditions qu'une fonction simple en enrichissant également le contexte de vérification de type de l'expression avec la liaison du nom de la fonction avec son type déclaré:

(FUNREC) si $\Gamma[x_1 : t_1; \ldots; x_n : t_n; x : t_1$ `*` $\ldots$ `*` t_n `->` $t] \vdash_{\text{Expr}} e : t$
alors $\Gamma \vdash_{\text{Dec}}$ (`FUN REC` x t `[`x_1`:`t_1`,` $\ldots$ `,`x_n`:`t_n`]` e) : $\Gamma[x : t_1$ `*` $\ldots$ `*` t_n `->` $t]$

1.2.4 Suites de commandes

Pour vérifier la correction d'une suite de commandes vis-à-vis du typage, on examine, récursivement, les éléments de la suite les uns après les autres, dans l'ordre donné par la suite. L'analyse se termine lorsque l'on atteint la *suite vide* que l'on note ε. On note d`;`cs une suite qui commence pas la déclaration d et se poursuit par la suite cs, et s`;`cs la suite qui commence par l'instruction s et se poursuit par la suite cs. Par abus de notation, la suite x`;`ε est la suite qui contient le seul élément x.

Lorsque la suite commence par une déclaration et que celle-ci est bien typée, on poursuit la vérification sur le reste de la suite avec le contexte obtenu après vérification de la déclaration de tête:

(DECS) si $d \in$ Dec, si $\Gamma \vdash_{\text{Dec}} d : \Gamma'$ et si $\Gamma' \vdash_{\text{Cmds}} cs$: *void* alors $\Gamma \vdash_{\text{Cmds}}$ (d`;` cs) : *void*.

Lorsque la suite commence par une instruction, on vérifie le bon typage de celle-ci avant de passer à l'ananlyse de la fin de la suite:

(STATS) si $s \in$ Stat, si $\Gamma \vdash_{\text{Stat}} s$: *void* et si si $\Gamma' \vdash_{\text{Cmds}} cs$: *void* alors $\Gamma \vdash_{\text{Cmds}}$ (s`;` cs) : *void*.

La suite vide est de type *void*:

(END) $\Gamma \vdash_{\text{Cmds}} \varepsilon$: *void*.

1.2.5 Programmes

Un programme est correctement typé si la suite de commandes qui le compose est correctement typée, dans le contexte initial Γ_0:

(PROG) si $\Gamma_0 \vdash_{\text{Cmds}} cs$: *void* alors $\vdash$ `[`cs`]` : *void*

1.3 Sémantique

La sémantique d'un langage de programmation doit définir quelle est la valeur ou l'effet attendu de l'exécution des programmes de ce langage. Pour cela, la sémantique définit le comportement dynamique des différents composants d'un programme. Comme les règles de typage, elle est définie par *cas de construction* des programmes.

Valeurs et environnements Autre analogie avec le typage, on définit la sémantique comme une relation entre un *contexte d'évaluation*, une construction syntaxique et une valeur ou un effet. Mais ici, les contextes d'évaluation associent une *valeur* aux identificateurs et non plus simplement un type.

Les valeurs manipulées lors de l'exécution d'un programme de *APS0* sont de trois sortes:

- les *valeurs immédiates* que nous pourrons limiter pour *APS0* au seul ensemble des entiers naturels. Dans la sémantique, nous ne distinguons pas valeurs numériques des valeurs booléennes: la séparation de leur usage ayant été garantie par le typage.

- les valeurs que nous devons associer aux fonctions et que nous appelons des *fermetures*. Nous motivons et définissons les fermetures un peu plus loin.

- les valeurs associées aux définitions récursives qui ne seront pas de simples fermetures, mais des fonctions de construction d'une fermeture.

Appelons N l'ensemble des entiers naturels (valeurs immédiates), F l'ensemble des fermertures et FR l'ensemble des fermetures récursives. On définit l'ensemble V des valeurs sémantiques pour *APS0* comme la *somme disjointe* des trois ensembles N, F et FR. On écrit $V = N \oplus F \oplus FR$.

Une somme (ou union) disjointe, comme son nom l'indique, réuni en un seul ensemble les éléments de plusieurs ensembles, mais, à la différence de l'union simple, on sait, dans une somme disjointe distinguer l'origine des éléments de la somme. Intuitivement, dans une somme disjointe, les éléments conservent une *marque* de leur origine. Formellement, cette marque est représentée par une *injection canonique*. Par exemple, pour l'ensemble V des valeurs, nous avons trois injections canoniques notées inN, inF et $inFR$ dont les domaines sont, respectivement, N, F et FR et qui partagent le codomaine V. Ainsi, tous les élements de V peuvent s'écrire sous l'une des formes: $inN(n)$, $inF(f)$ ou $inFR(r)$ avec $n \in N$, $f \in F$ et $r \in FR$.

L'ensemble E des environnements est l'ensemble des fonctions partielles qui associent une valeur à un identificateur. On écrit $E = \mathsf{ident} \to V$.

Comme les contextes de typage, les environnements évolueront au fur et à mesure de l'évaluation. Par exemple, une déclaration ajoute une liaison entre identificateur et valeur à l'environnement. Si ρ est un environnment, x, un identificateur et v une valeur, on note $\rho[x = v]$ l'extension de l'environnement ρ avec la nouvelle liaison de x à v. La notation $\rho[x = v]$ désigne la fonction de E telle que $\rho[x = v](x) = v$ et $\rho[x = v](y) = \rho(y)$ lorsque x et y sont des symboles différents.

Flux de sortie Un autre élément à *modéliser* pour définir la sémantique des programmes de *APS0* est l'effet de l'instruction d'affichage: l'envoi ou l'ajout d'une valeur dans un *flux de sortie*. Abstraitement, un flux de sortie est simplement une suite de valeurs. On appelle O l'ensemble des flux de sortie. On écrit $O = N^*$. Si ω est un flux de sortie, et n un entier, on note $n \cdot \omega$ l'ajout de n au flux de sortie.

Valeurs fonctionnelles: les fermetures Une *fonction* dans *APS0* est donnée par la liste de ses paramètres formels et une expression que l'on appelle le *corps* de la fonction. La valeur associée à une fonction doit permettre l'application futur de celle-ci; c'est-à-dire l'évaluation du corps de la fonction dans un contexte d'évaluation où les paramètres formels de la fonction sont liés aux valeurs des paramètres d'appel.

Le corps d'une fonction fait mention de ses paramètres formels, mais il peut également faire référence à d'autres constantes ou fonctions définies dans le programme et qui devront être présentes dans le contexte d'évaluation lors de l'appel (*i.e.* application) des fonctions. Pour ces dernières, il y a deux manières d'envisager les choses:

- ou bien, à chaque appel de fonction, on utilise le contexte d'évaluation présent au moment de l'appel, on parle alors de *liaison dynamique*;

- ou bien, on fige, au moment de la création de la fonction (par exemple, de sa déclaration), le contexte d'évaluation présent à ce moment et on utilise ce contexte à chaque appel de la fonction, on parle alors de *liaison statique*.

Nous utiliserons pour *APS0*, et ses successeurs, la liaison statique.

Pour la sémantique d'APS0, et de ses successeurs, une fermeture, qui est la *valeur* donnée aux fonctions, sera composée d'une expression (le corps de la fonction) et d'une fonction (sémantique) de construction d'environnement. Une fermeture a donc la forme d'un couple (e, r) où e est le corps de la fonction (*APS0*) et r, une fonction (sémantique) de construction d'environnement.

Si la fonction dont on veut donner la valeur a pour paramètres $x_1, \ldots, x_n$ et que le contexte courant est ρ, la fonction de construction de l'environnement est une fonction qui attend n valeurs $(v_1, \ldots, v_n)$ et qui ajoute à ρ les liaisons $x_1 = v_1, \ldots, x_n = v_n$. On écrit: $\lambda v_1, \ldots, v_n.\rho[x_1 = v_1; \ldots; x_n = v_n]$.

L'ensemble des femetures est donc l'ensemble des couples formés d'une expression et d'une fonction des (suites de) valeurs dans les environnements. On pose: $F = \textsc{Expr} \times (V^* \to E)$.

La valeur de la fonction de paramètres formels $x_1, \ldots, x_n$ et de corps e, créée dans le contexte ρ s'écrit

$$inF(e, \lambda v_1, \ldots, v_n.\rho[x_1 = v_1, \ldots, x_n = v_n])$$

Dans le cas où la fonction est récursive, l'expression qui constitue le corps de la fonction fait usage du nom de la fonction elle-même. Par tant, l'environnement qu'il faudra construire au moment de l'application de la fonction devra contenir une référence à la fonction elle-même. Il aura la forme

$$\rho[x_1 = v_1; \ldots; x_n = v_n][x = f]$$

où $x_1, \ldots, x_n$ sont les noms des paramètres formels de la fonction et x, le nom de la fonction elle-même; $v_1, \ldots, v_n$ seront les valeurs des paramètres d'appel et f, la valeur (fermeture) de la fonction elle-même.

On ne peut ici déterminer f. En effet, il faut une fermeture de la forme

$$inF(e, \lambda v_1, \ldots, v_n.\rho[x_1 = v_1; \ldots; x_n = v_n][x = f])$$

avec

$$f = inF(e, \lambda v_1, \ldots, v_n.\rho[x_1 = v_1; \ldots; x_n = v_n][x = f])$$

Ce qui donne une définition cyclique.

Pour ne pas avoir de définition de fermeture cyclique, nous retarderons l'auto-référence jusqu'au moment de l'application de la fonction. Une fermeture récursive sera donc une fonction qui attend une fermeture récursive pour produire une fermeture. La valeur d'une fonction récursive de nom x, de paramètres $x_1, \ldots, x_n$, de corps e, définie dans le contexte ρ s'écrit

$$inFR(\lambda f.inF(e, \lambda v_1, \ldots, v_n.\rho[x_1 = v_1; \ldots; x_n = v_n][x = f]))$$

Ainsi définie, une fermeture récursive est donc une fonction qui attend une valeur (qui sera, en fait, toujours une fermeture récursive) pour donner une fermeture. On pose $FR = V \to F$.

Remarque: poser $FR = FR \to F$ soulève quelques difficultés théoriques que nous préférons éviter ici.

Domaines sémantiques Si l'on résume la collection d'ensembles décrits ci-dessus, on obtient le tableau

Valeurs	$V = N \oplus F \oplus FR$
Valeurs immédiates	N
Fermetures	$F = \textsc{Expr} \times (V^* \to E)$
Fermetures récursives	$FR = V \to F$
Environnement	$E = \mathsf{ident} \to V$
Flux de sortie	$O = N^*$

Il conviendrait de justifier le bien fondé des quatre égalités:

$$\begin{array}{lcl} V & = & N \oplus F \oplus FR \\ F & = & \textsc{Expr} \times (V^* \to E) \\ FR & = & V \to F \\ E & = & \mathsf{ident} \to V \end{array}$$

qui sont *mutellement récursives.*

Contextes d'évaluation Un contexte d'évaluation est un couple formé d'un environnement (fonction dans E) et d'un flux de sortie.

Fonctions utiles La possibilité de faire réaliser des calculs par les programmes de *APS0*, comme pour tout autre langage de programmation, repose sur l'existence de *fonctions primitives* que les programmes peuvent invoquer. Dans les «vrais langages», ceux qui sont destinés à tourner sur des «vraies machines», ces fonctions primitives sont *implantées* dans les micro-processeurs par des circuits réalisant les opérations arithmétiques, pour les plus simples, ou du code assembleur pour les plus complexes. Nous ne descendrons pas jusqu'à ce niveau de détail et nous contenterons de supposer l'existence d'une fonction qui saura associer à chaque symbole d'opération primitive un calcul sur les valeurs entières. Appelons π cette fonction et posons:

$$
\begin{array}{rcll}
\pi(\texttt{not})(0) & = & 1 & \\
\pi(\texttt{not})(1) & = & 0 & \\
\pi(\texttt{and})(0,n) & = & 0 & \\
\pi(\texttt{and})(1,n) & = & n & \\
\pi(\texttt{or})(1,n) & = & 1 & \\
\pi(\texttt{or})(0,n) & = & n & \\
\pi(\texttt{eq})(n_1,n_2) & = & 1 & \text{si } n_1 = n_2 \\
 & = & 0 & \text{sinon} \\
\pi(\texttt{lt})(n_1,n_2) & = & 1 & \text{si } n_1 < n_2 \\
 & = & 0 & \text{sinon} \\
\pi(\texttt{add})(n_1,n_2) & = & n_1 + n_2 & \\
\pi(\texttt{sub})(n_1,n_2) & = & n_1 - n_2 & \\
\pi(\texttt{mul})(n_1,n_2) & = & n_1 \times n_2 & \\
\pi(\texttt{div})(n_1,n_2) & = & n_1 \div n_2 &
\end{array}
$$

Enfin, le code des programmes fera mention de constantes numériques: les unités lexicales de l'ensemble num. Les calculs ne sont pas effectués directement sur les symboles de ces constantes, mais sur les valeurs immédiates qu'ils représentent. Pour les «vraies machines» ces valeurs immédiates sont les codages binaires des nombres, pour nous ce sera plus simplement les éléments de N. Il nous faut donc un moyen de passer des éléments de num (suites de caractères) à ceux de N. Pour ce, on se donne une fonction ν telle que, par exemple $\nu(\texttt{42}) = 42$.

Relations sémantiques On retrouve pour la sémantique cinq relations liant les contextes d'évaluation et les constructions syntaxiques à leur résultat. Nous utiliserons les mêmes symboles que pour le typage, mais leurs domaines et notation seront différents.

1. L'évaluation d'un programme *APS0* produit un flux de sortie, la relation sémantique qui la définit associe un programme à un élément de O.

   Relation $\vdash$ de domaine $\textsc{Prog} \times O$, on écrit $\vdash p \rightsquigarrow \omega$.

2. L'évaluation d'une déclaration produit un nouvel environnement, la relation sémantique qui la définit associe un environnement et une déclaration à un nouvele environnement.

   Relation $\vdash_{\textsc{Dec}}$ de domaine $E \times \textsc{Dec} \times E$, on écrit $\rho \vdash_{\textsc{Dec}} d \rightsquigarrow \rho'$.

3. L'évaluation de l'instruction produit un nouveau flux de sortie en utilisant la valeur obtenue par évaluation d'une expression, sa relation sémantique associe un environnement, un flux de sortie et une instruction à un nouveau flux de sortie.

   Relation $\vdash_{\textsc{Stat}}$ dans $E \times O \times \textsc{Stat} \times O$, on écrit $\rho, \omega \vdash_{\textsc{Stat}} s \rightsquigarrow \omega$.

4. L'évaluation d'une expression produit une valeur, sa relation sémantique associe un environnement et une expression à une valeur.

   Relation $\vdash_{\textsc{Expr}}$ de domaine $E \times \textsc{Expr} \times V$, on écrit $\rho \vdash_{\textsc{Expr}} e \rightsquigarrow v$.

5. Enfin, l'évaluation d'une suite de commandes produit un nouveau flux de sortie, sa relation sémantique associe un environnement, un flux de sortie et une suite de commandes à un nouveau flux de sortie. Comme pour le typage, on évalue des suites terminées par la commande vide ε.

   Relation $\vdash_{\text{CMDS}}$ de domaine $E \times O \times \text{CMDS}_\varepsilon \times O$, on écrit $\rho, \omega \vdash_{\text{CMDS}} cs \rightsquigarrow \omega'$.

1.3.1 Expressions

Les valeurs des constantes booléennes `true` et `false` sont, respectivement les valeurs 1 et 0.

(TRUE) $\rho \vdash_{\text{EXPR}} \texttt{true} \rightsquigarrow inN(1)$

(FALSE) $\rho \vdash_{\text{EXPR}} \texttt{false} \rightsquigarrow inN(0)$

La valeur d'une constante numérique est donnée par la fonction ν:

(NUM) si $n \in \mathsf{num}$ alors $\rho \vdash_{\text{EXPR}} n \rightsquigarrow inN(\nu(n))$

La valeur d'un identificateur est celle donnée par l'environnement:

(ID) si $x \in \mathsf{ident}$ et $\rho(x) = v$ alors $\rho \vdash_{\text{EXPR}} x \rightsquigarrow v$

Notez que la relation n'est pas définie si $\rho(x)$ ne l'est pas.

La valeur de l'application d'une fonction primitive est obtenue par appel aux fonctions fournies par la fonction π, après évaluation des arguments appliqués:

(PRIM) si $x \in \mathsf{oprim}$, si $\rho \vdash_{\text{EXPR}} e_1 \rightsquigarrow inN(n_1)$, ..., si $\rho \vdash_{\text{EXPR}} e_k \rightsquigarrow inN(n_k)$ et si $\pi(x)(n_1, \ldots, n_k) = n$
alors $\rho \vdash_{\text{EXPR}} (x\ e_1 \ldots e_k) \rightsquigarrow inN(n)$

La valeur d'une expression alternative dépend de la valeur de son premier argument:

(IF1) si $\rho \vdash_{\text{EXPR}} e_1 \rightsquigarrow inN(1)$ et si $\rho \vdash_{\text{EXPR}} e_2 \rightsquigarrow v$ alors $\rho \vdash_{\text{EXPR}} (\texttt{if}\ e_1\ e_2\ e_3) \rightsquigarrow v$

(IF0) si $\rho \vdash_{\text{EXPR}} e_1 \rightsquigarrow inN(0)$ et si $\rho \vdash_{\text{EXPR}} e_3 \rightsquigarrow v$ alors $\rho \vdash_{\text{EXPR}} (\texttt{if}\ e_1\ e_2\ e_3) \rightsquigarrow v$

Notez comment, dans une expression alternative seuls deux arguments sont évalués, et non trois.

La valeur d'une abstraction fonctionnelle est une fermeture capturant l'environnement courant

(ABS) $\rho \vdash_{\text{EXPR}} [x_1 : t_1, \ldots, x_n : t_n] e \rightsquigarrow inF(e, \lambda v_1 \ldots v_n . \rho[x_1 = v_1; \ldots; x_n = v_n])$

La valeur de l'application d'une expression (dont la valeur doit être une fermeture) à d'autres expressions est obtenue, après évaluation de ces dernières, par évaluation du premier terme de la fermeture (corps de la fonction) dans le contexte construit par extension du contexte capturé dans la fermeture avec les valeurs obtenues pour les arguments d'appel:

(APP) si $\rho \vdash_{\text{EXPR}} e \rightsquigarrow inF(e', r)$, si $\rho \vdash_{\text{EXPR}} e_1 \rightsquigarrow v_1$, ..., si $\rho \vdash_{\text{EXPR}} e_n \rightsquigarrow v_n$
et si $r(v_1, \ldots, v_n) \vdash_{\text{EXPR}} e' \rightsquigarrow v$
alors $\rho \vdash (e\ e_1 \ldots e_n) \rightsquigarrow v$

Si la valeur de l'expression en position de fonction est une fermeture récursive, on applique cette fermeture à elle-même avant de procéder comme ci-dessus:

(APPR) si $\rho \vdash_{\text{EXPR}} e \rightsquigarrow inFR(\varphi)$, si $\varphi(inFR(\varphi)) = inF(e', r)$,
si $\rho \vdash_{\text{EXPR}} e_1 \rightsquigarrow v_1$, ..., si $\rho \vdash_{\text{EXPR}} e_n \rightsquigarrow v_n$
et si $r(v_1, \ldots, v_n) \vdash_{\text{EXPR}} e' \rightsquigarrow v$
alors $\rho \vdash (e\ e_1 \ldots e_n) \rightsquigarrow v$

L'application $\varphi(inFR(\varphi))$ réalise la liaison de $inFR(\varphi)$ avec le nom de la fonction récursive (voir ci-dessous: déclaration de fonction récursive).

1.3.2 Instruction

L'évaluation de l'instruction d'affichage ajoute un entier au flux de sortie:

(ECHO) si $\rho, \omega \vdash_{\text{EXPR}} e \rightsquigarrow inN(n)$ alors $\rho, \omega \vdash_{\text{STAT}} \texttt{ECHO}\ e \rightsquigarrow (n \cdot \omega)$

1.3.3 Déclaration

L'évaluation d'une déclaration de constante ajoute à l'environnement la liaison entre le nom de la constante et la valeur obtenue par évaluation de l'expression qui la définit:

(CONST) si $\rho \vdash_{\text{EXPR}} e \rightsquigarrow v$ alors $\rho \vdash_{\text{DEC}} (\texttt{CONST}\ x\ t\ e) \rightsquigarrow \rho[x = v]$

L'évaluation de la déclaration d'une fonction (non récursive) ajoute à l'environnement la liaison entre le nom de la fonction et la fermeture qui représente cette fonction:

(FUN) $\rho \vdash_{\text{DEC}} (\texttt{FUN}\ x\ t\ \texttt{[}x_1\texttt{:}t_1\texttt{,}\ \ldots\ \texttt{,}x_n\texttt{:}t_n\texttt{]}\ e) \rightsquigarrow \rho[x = inF(e, \lambda v_1 \ldots v_n.\rho[x_1 = v_1; \ldots; x_n = v_n])$

L'évaluation de la déclaration d'une fonction récursive ajoute à l'environnement une liaison entre le nom de la fonction et la fermeture récursive qui la représente:

(FUNREC) $\rho \vdash_{\text{DEC}} (\texttt{FUN REC}\ x\ t\ \texttt{[}x_1\texttt{:}t_1\texttt{,}\ \ldots\ \texttt{,}x_n\texttt{:}t_n\texttt{]}\ e)$
$\rightsquigarrow \rho[x = inFR(\lambda f.inF(e, \lambda v_1 \ldots v_n.\rho[x_1 = v_1; \ldots; x_n = v_n][x = f])$

1.3.4 Suite de commandes

On distingue le cas des suites commençant par une déclaration de celles commençant par une instruction.

Si la suite commence par une déclaration, l'évaluation de la suite est obtenue par l'évaluation du reste de la suite dans un environnement produit par la déclaration:

(DECS) si $\rho, \omega \vdash_{\text{DEC}} d \rightsquigarrow \rho'$ et si $\rho', \omega \vdash_{\text{CMDS}} cs \rightsquigarrow \omega'$ alors $\rho, \omega \vdash_{\text{CMDS}} (d\texttt{;}\ \ cs) \rightsquigarrow \omega'$

Si la suite commence par une instruction, l'évaluation de la suite est obtenue par l'évaluation du reste de la suite dans un contexte où le flux de sortie a été modifié par l'évaluation de l'instruction:

(STATS) si $\rho, \omega \vdash_{\text{STAT}} s \rightsquigarrow \omega'$ et si $\rho, \omega' \vdash_{\text{CMDS}} cs \rightsquigarrow \omega''$ alors $\rho, \omega \vdash_{\text{CMDS}} (s\texttt{;}\ \ cs) \rightsquigarrow \omega''$

L'évaluation de la suite vide laisse inchangé le flux de sortie:

(END) $\rho, \omega \vdash_{\text{CMDS}} \varepsilon \rightsquigarrow \omega$

1.3.5 Programme

L'évaluation d'un programme est l'évaluation de la suite de commandes (complétée par l'instruction vide ε) qui le constitue avec un contexte initial vide noté $\emptyset, \emptyset$:

(PROG) si $\emptyset, \emptyset \vdash_{\text{CMDS}} (cs\texttt{;}\varepsilon) \rightsquigarrow \omega$ alors $\vdash \texttt{[}cs\texttt{]} \rightsquigarrow \omega$