Sémantique des langages de programmation

Jacques Malenfant
professeur des universités

Table des matières

Chapitre 1

Introduction

L'étude des langages de programmation est l'un des domaines importants et l'un des plus anciens de la science informatique. Comme tous les domaines en maturation, il se formalise de plus en plus. Loin est l'époque de la conception des premières versions de Fortran et de la mise en oeuvre de son premier compilateur où tout n'était que travail d'orfèvre. Aujourd'hui, de la conception à l'utilisation d'un langage de programmation, de nombreux outils formels et pratiques jouent un rôle crucial. Cela va des automates d'états finis et des grammaires jusqu'aux implantations formellement certifiées, en passant par les générateurs d'analyseurs comme Lex et Yacc.

La tendance à la formalisation est une garantie de qualité et elle sert de base au développement des outils pratiques, qui eux-mêmes ont fait gagner des milliers d'heures de travail. Non seulement la mise en oeuvre des langages de programmation en est-elle largement simplifiée, mais la réalisation d'applications pratiques dans ces langages gagnent en efficacité par l'utilisation de compilateurs plus fiables, produisant plus rapidement du code de meilleure qualité. En ce sens, les techniques formelles font partie du bagage indispensable dans la recherche sur les langages de programmation.

L'expérience montre cependant que cette même tendance à la formalisation a créé une fracture entre les concepteurs de langages et le monde de la recherche dans le domaine des langages de programmation. Le concepteur de langage est souvent un chercheur qui se situe au plus près des utilisateurs de langages de programmation. Rarement conçoit-il des langages réellement totalement nouveaux ; il cherche plutôt à proposer de nouvelles abstractions de programmation qui permettent d'exprimer plus facilement la solution aux problèmes courants de ces utilisateurs. Bien que capables d'exprimer leurs nouvelles abstractions de façon informelles, les concepteurs se butent à plusieurs problèmes :

1. Comment raisonner sur le comportement des programmes utilisant ces nouvelles abstractions ?
2. Comment s'assurer de l'exactitude de leurs propositions ?
3. Comment comparer différentes propositions de façon à en obtenir des conclusions claires et définitives ?
4. Comment évaluer la réalisabilité de la mise en oeuvre de ces nouvelles abstractions ?
5. Comment communiquer de façon précise et non-ambigue la sémantique de ces nouvelles abstractions aux utilisateurs potentiels ? et
6. Comment démontrer l'intérêt, le potentiel et la réalisabilité de ces nouvelles abstractions à la communauté de recherche ?

Après la formalisation de tout ce qui correspond à la partie frontale du compilateur et qui concerne donc la syntaxe et la sémantique statique[1] des langages[2], la formalisation des aspects sémantiques prend un essor rapide depuis bientôt une décennie. Les principales approches formelles disponibles aujourd'hui pour la définition de la sémantique des langages et relativement largement connues sont les sémantiques opérationnelles structurelles, les sémantiques dénotationnelles et les sémantiques axiomatiques. Ces techniques donnent effectivement des réponses aux problèmes évoqués plus haut, mais elles colportent les mêmes désavantages que toutes les méthodes formelles : difficultés d'apprentissage, opacité des définitions et limites intrinsèques[3].

Malgré tout, nous sommes profondément convaincus que les approches formelles sont devenues aujourd'hui indispensables. Le présent cours vise donc un public de concepteurs de langages de programmation et a pour objectif de leur permettre de s'approprier l'une de ces approches, la sémantique dénotationnelle. La trame de fond du cours évitera d'aborder les techniques formelles sous un angle purement mathématique. Notre objectif n'est pas de faire du concepteur de langage un sémanticien de haut vol. Il s'agit plutôt de lui permettre d'exprimer dans le langage dénotationnel la sémantique de ses propositions. Il pourra ensuite, en collaboration avec le sémanticien, mettre en oeuvre les techniques de raisonnement formels pour prouver des propriétés sur les programmes.

Nous aborderons donc la sémantique formelle sous un angle pratique. Par rapport à l'approche classique de démonstration d'un langage, qui est la réalisation d'une implantation prototype sous la forme d'un interprète, nous faisons même le pari que la sémantique dénotationnelle ne présente pas une difficulté insurmontable. En fait, elle peut apparaître comme une technique particulière de définition d'interprètes qui, avec des garde-fous appropriés, présenteront les propriétés nécessaires et suffisantes pour être des sémantiques dénotationnelles. En corollaire, nous mettrons l'emphase sur l'obtention de sémantiques formelles *exécutables*, qui fourniront donc, en passant, une implantation prototype de même nature que les interprètes classiques.

Après un bref rappel des concepts des langages de programmation et leur réalisation dans le langage de programmation Scheme, nous introduirons la sémantique dénotationelle en commençant par le λ-calcul. Nous verrons les techniques de base de la sémantique dénotationnelle pour aborder ensuite les domaines et glisser sur les aspects plus théoriques donnant une justification à l'utilisation du formalisme. Enfin, nous développerons une bagage d'outils classiques (le terme à la mode serait «*patterns*») pour définir la sémantique de concepts régulièrement rencontrés dans la plupart des langages. Nous terminerons enfin en appliquant l'approche dénotationnelle à la définition de la sémantique des concepts de la programmation par objets.

[1] C'est-à-dire ce qui concerne l'observance de règles comme la déclaration des variables avant leur utilisation ou la congruence entre la liste des paramèters formels et la liste des paramètres réels lors de l'appel d'une fonction ou d'une procédure.

[2] Pensons aux automates d'états finis, aux grammaires hors contexte et aux grammaires attribuées.

[3] Il reste de la recherche à faire dans le domaine des approches formelles pour la définition de la sémantique des langages de programmation...

Chapitre 2

Concepts des langages de programmation en Scheme

Avant d'aborder la sémantique des langages de programmation, ce chapitre offre un rappel des principaux concepts des langages. La présentation illustre ces concepts à l'aide du langage de programmation fonctionelle Scheme. Deux raisons justifient ce choix. D'une part, Scheme, comme tous les langages fonctionnels, est basé sur le λ-calcul et nous permet ainsi d'appréhender de façon pratique et progressive nombre de concepts et de phénomènes que nous reverrons plus formellement ensuite en λ-calcul. D'autre part, également parce qu'il est fonctionnel, Scheme nous offre un véhicule de mise en oeuvre de nos sémantiques dénotationnelles ; il est donc utile d'en rappeler les grandes lignes ici.

2.1 Concepts des langages de programmation

Sous des habillages très variés, les langages de programmation sont en réalité fondés sur un ensemble de concepts relativement petit. Très tôt dans l'apprentissage des langages de programmation, l'informaticien perspicace se rend vite compte que d'un langage à l'autre, bien que chaque langage ait sa personnalité propre en lien avec son domaine d'application, plusieurs langages donnent l'impression de ne présenter, en grande partie, que des variations plus ou moins similaires sur un thème connu. Cette impression est particulièrement prégnante lorsqu'on reste dans le même paradigme de programmation ; il y a une grande similarité entre Pascal et C d'une part (perogrammation impérative) et entre Scheme et ML d'autre part (programmation fonctionnelle).

L'informaticien chevronné saura reconnaître ces similitudes comme des réalisations plus ou moins différentes des mêmes concepts de base : constantes, variables, expressions, énoncés, structures de contrôle, abstractions fonctionnelles et procédurales, etc. Le concepteur de langages sait qu'un nouveau langage ne fait souvent que greffer quelques concepts nouveaux autour de ce noyau dur. Dans cette section, nous faisons un bref rappel des principaux concepts des langages de programmation. Pour plus de détails, le lecteur est référé aux livres classiques de ce domaine [Set96].

Un langage de programmation est essentiellement un véhicule permettant d'exprimer la solution à un problème d'une manière suffisamment précise pour en permettre l'exécution sur un ordinateur. Il vise à fournir un modèle de calcul de plus haut niveau, plus facile à comprendre que celui du processeur et du langage d'assemblage. Cette facilité de compréhension

$$
\begin{array}{lcl}
p & ::= & e \\
e & ::= & n \mid e + e \mid e - e \mid e * e \mid e \,/\, e \\
n & ::= & nc \mid c \\
c & ::= & 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9
\end{array}
$$

FIG. 2.1 – Grammaire des expression arithmétiques

vise à permettre :

1. l'écriture de programmes corrects,
2. l'entretien de ce logiciel par des modifications successives visant à corriger les fautes de programmation ou pour accomoder de nouveaux besoins des utilisateurs, et
3. la réutilisation de portions de programmes pour réaliser plus vite et de façon moins coûteuse de nouveaux programmes.

Un langage de programmation se définit en quatre grandes parties :

- Son vocabulaire, dont la reconnaissance est confiée à l'analyseur lexicographique.
- Sa syntaxe, dont la reconnaissance est confiée à l'analyseur syntaxique.
- Ses règles sémantiques, dont la vérification de leur observance est confiée à l'analyseur sémantique.
- Sa signification, dont la réalisation est obtenue par traduction du code de haut niveau vers du code machine.

Le vocabulaire d'un langage est constitué de constantes, de mots-clés, de symboles de ponctuation, d'identificateurs, etc. Leur description est souvent donnée sous forme d'expressions régulières. Des outils comme `lex` et `flex` transforment automatiquement une spécification des unités lexicales d'un langage sous formes d'expressions régulières en un analyseur lexicographique.

La syntaxe d'un langage définit la forme des énoncés et des expressions. Formellement, sa description est souvent donnée sous forme d'une grammaire dite indépendante du contexte. La notation la plus couramment utilisée pour définir une grammaire est la notation de Backus-Naur, dont un exemple est donné à la figure 2.1. Des outils comme `yacc` et `bison` transforment automatiquement une spécification de la syntaxe d'un langage sous forme d'une grammaire en un analyseur syntaxique.

La sémantique statique d'un langage exprime un certain nombre de règles que doivent observer les programmes pour être considérés corrects. Typiquement, la congruence entre paramètres réels et paramètres formels d'une fonction est une de ces règles (même nombre d'arguments à l'appel de la fonction qu'elle a de paramètres dans sa définition). De même, la concordance des types fait partie de la sémantique statique. Il existe une grande variété de règles de sémantique statique, et par conséquent plusieurs outils sont utilisés pour la définir. Un des plus courants sont les grammaires attribuées.

La sémantique dynamique d'un langage exprime la signification en terme de résultat du programme lors de son exécution. Il existe plusieurs façons d'exprimer cette sémantique, les plus courantes étant des descriptions en langage naturel et des implantations concrètes (compilateur et/ou interprète). Mais de plus en plus, on voit apparaître des descriptions formelles, dont la sémantique opérationnelle structurelle, la sémantique dénotationnelle et la sémantique axiomatique.

Un paradigme de programmation s'articule autour d'un modèle d'exécution des programmes et propose tout à la fois une manière de raisonner sur cette exécution et les moyens pour spécifier un calcul (au sens large) correct. Par exemple, le paradigme impératif repose sur un modèle d'exécution où l'ordinateur est vu comme une mémoire ayant un état courant et où un calcul est essentiellement une séquence d'instructions faisant progresser l'ordinateur d'un état à un autre. L'état initial contient (normalement) une représentation du problème à résoudre, alors que l'état final est présumé contenir une description de la solution recherchée. Aujourd'hui, lorsqu'on parle de programmation impérative, on entend généralement programmation structurée. La programmation structurée a été inventée pour faciliter le raisonnnement sur les programmes impératifs en forçant une structuration adéquate du flôt d'exécution des programmes. En particulier, les structures de contrôle dans la syntaxe de ces langages n'ont généralement qu'un point d'entrée et un point de sortie, ce qui facilite grandement le raisonnement sur l'ordre d'exécution des instructions.

Pour l'essentiel, nous supposons connus les principaux éléments constituant un langage :

- Constantes : nombres, caractères, chaînes de caractères, pointeurs.
- Types de données structurés : tableaux, structures, listes, etc.
- Expressions : arithmétiques, booléennes, etc. Elles sont exécutées pour leur résultat.
- Fonctions : abstraction de l'expression, la fonction paramétrise et donne un nom à une expression.
- Énoncés : affectation, boucles, alternatives, etc. Ils sont exécutés pour leurs effets sur les données.
- Procédures : abstraction de l'énoncé, la procédure paramétrise et donne un nom à une (séquence d') énoncé(s).

La notion de fonction et de procédure introduit une distinction entre leur définition et leur exécution. La définition d'une fonction ou d'un procédure se compose d'un nom, d'une liste de paramètres formels, d'un type de résultat pour la fonction et d'un corps. L'appel d'une fonction (resp. d'une procédure) est une expression (resp. énoncé) particulière se composant d'un nom et d'une séquence de paramètres réels ; l'appel se réalise par l'activation de la fonction (resp. procédure) de même nom. L'activation comporte trois phases distinctes : le passage des paramètres, l'exécution du corps, puis le retour au contexte d'appel qui inclut le retour du résultat dans le cas de la fonction. On distingue trois méthodes principales pour le passage de paramètres : l'appel par valeur, l'appel par référence et l'appel par nom. La première, appel par valeur, passe une copie de la valeur à la fonction ou procédure appelée alors que la seconde, appel par référence, passe l'adresse de l'emplacement mémoire où se trouve la valeur du paramètre réel. La troisième est plus complexe, puisqu'elle passe l'expression à exécuter pour obtenir la valeur du paramètre qui est généralement évaluée au besoin ; la principale difficulté est alors d'exécuter cette expression non dans le contexte de la fonction (ou procédure) mais plutôt dans le contexte de son appel.

Par ailleurs, trois notions auront une importance majeure dans le reste du cours. Les deux premières sont la portée et la durée de vie des identificateurs. La portée des identificateurs et les constructions syntaxiques (fonctions, procédures, blocs, modules, objets, etc.) permettant de la limiter sont à l'origine de plusieurs des problèmes sémantiques que nous allons discuter. La fonction et la procédure paramétrise un calcul et les identificateurs utilisés comme paramètres formels et comme variables locales ont une portée qui se limite à l'intérieur de la fonction ou de la procédure. Leur durée de vie diffère cependant selon les langages. En Fortran IV, la durée de vie des variables locales est l'exécution entière du programme. En C et en Pascal, elle se limite à la durée de l'activation de la fonction qui les déclare et une nouvelle incarnation de chaque

variable est créée pour chaque activation de la procédure (on dit qu'elles ont une durée de vie dynamique). Cela permet la récursivité. Les langages comme Scheme et Smalltalk permettent la création de *fermetures* capables de capturer les variables locales et pouvant être retournées comme résultat de la fonction. Ainsi, les variables ont une incarnation par activation mais aussi une durée de vie illimitée. Ces phénomènes induisent des sémantiques différentes.

La troisième notion est celle de syntaxe abstraite. On distingue deux types de syntaxe pour décrire un programme. La syntaxe concrète définit l'agencement des unités lexicales qui forment les énoncés et les expressions du langage. La syntaxe concrète sert uniquement à analyser les programmes pour déterminer s'ils sont correctement construits. La syntaxe abstraite ne s'intéresse qu'à la structure du programme et permet de raisonner sur cette dernière sans s'attacher à une forme extérieure précise. Par exemple, comment une expression est formée de sous-expressions, celles-ci étant formées de l'application d'un opérateur sur des opérandes, ou d'un appel de fonction.[1] La syntaxe abstraite se définit sous forme d'arbres et elle ne sert qu'à exprimer la structure des programmes et le type des constructions utilisées.

De façon générale, on utilise la syntaxe concrète pour analyser les programmes, et au cours de l'analyse, on construit l'arbre de syntaxe abstraite représentant la structure du programme analysé. L'arbre de syntaxe abstraite est ensuite utilisé pour les phases subséquentes du traitement, c'est-à-dire l'analyse sémantique et la génération de code dans un compilateur. Lorsque l'on veut parler de l'ensemble des structures de programme que l'on peut écrire dans un langage, il est utile de parler de l'ensemble des arbres de syntaxe abstraite correctement formés. Pour définir l'ensemble de ces arbres, on utilise des grammaires abstraites.[2]

2.2 Le langage Scheme

Le langage Scheme provient d'une rationalisation du langage Lisp, et la référence faisant autorité demeure la norme de fait établie par le «*Revised*^{4} *Report on the Algorithmic Language Scheme*» [CR91]. Nous n'en présenterons ici que les grandes lignes, référant le lecteur à l'un des nombreux livres introduisant le langage [Cha96, par exemple, en français] pour son apprentissage, et à la norme de fait pour les détails d'utilisation.

2.2.1 Eléments de syntaxe

Les constantes de base de Scheme sont assez traditionelles : entiers, réels, caractères (par exemple `#\a`), les booléens (`#T`, `#F`), les chaînes de caractères (par exemple `"abc"`), et les symboles (identificateur précédé d'un symbole de citation «'» (en anglais *quote*), par exemple `'abc`). Le principal type de données structuré de Scheme est la liste. Une littéral de type liste s'écrit comme une suite d'éléments entre parenthèses le tout précédé d'une symbole de citation (pour la distinguer d'une expression, voir tout de suite après) :

```
'(1 2 3)
```

La liste vide, qui ne contient aucun élément, est une valeur particulière qui s'écrit `'()`.

La syntaxe de Scheme démontre une grande régularité. Pour l'essentiel, Scheme est un langage dont les programmes sont formés d'expressions d'application de fonction en notation

[1] S'il s'agit de raisonner sur l'effet sémantique d'un énoncé d'affectation, point n'est besoin de savoir que l'opérateur d'affectation du langage se présente extérieurement comme la séquence de caractères `:=`.

[2] Cela est d'ailleurs source de confusion entre syntaxe abstraite et syntaxe concrète, puisqu'on utilise aussi des grammaires pour définir cette dernière, mais alors il s'agit de grammaires concrètes.

préfixée. Une expression est donc constituée d'un opérateur suivi de ses paramètres réels, le tout entre parenthèses :

```
(+ 3 4)
```

La forme générale d'une expression Scheme est :

(E_1 E_2 ... E_n)

où l'expression E_1 représente un opérateur qui sera appliqué aux valeurs résultant de l'évaluation des expressions E_2 ... E_n. L'ordre d'évaluations des expressions E_1 E_2 ... E_n n'est pas spécifié, mais toutes doivent avoir été évaluées avant l'application de la valeur de E_1 à ses opérandes.

Cette syntaxe régulière est adoptée dans l'ensemble du langage, y compris dans les structures de contrôle. Pour certaines structures de contrôle cependant, on ne peut voir le symbole en position d'opérateur comme étant une expression à évaluer ; dans ce cas, on parle de formes spéciales. C'est le cas de la conditionnelle de Scheme, qui prend la forme :

(`if` P E_1 E_2)

où P est une expression prédicative (dont le résultat est un booléen). Si P s'évalue à vrai (`#T`), alors l'expression E_1 est évaluée et son résultat devient le résultat de la forme entière. Si P s'évalue à faux (`#F`), c'est alors E_2 qui est évaluée et son résultat qui devient le résultat de la forme. Notons que `if` n'est pas une fonction au sens de Scheme car ses «arguments» ne sont pas tous évalués (E_1 ou E_2 est évaluée, mais pas les deux).

La définition d'une nouvelle variable et sa liaison à une valeur sont réalisées par la forme `define`. Par exemple, la création d'une variable `x` de valeur 10 et d'une fonction `plus-grand` retournant le plus grand de ses éléments se réalisent par :

```
(define x 10)

  (define (plus-grand x y)
    (if (> x y)
      x
      y))
```

L'énoncé d'affectation se présente également comme une forme spéciale introduite par le symbole `set!` :

```
(set! x 20)
```

a pour effet de modifier la valeur de la variable `x` créée précédemment qui vaudra maintenant `20`. Notez que le caractère « `!`» (prononcez «bang») terminant le symbole `set!` est normalement utilisé en Scheme pour mettre l'emphase sur les effets de bords. Il est donc de bon aloi de terminer le nom d'une fonction qui fait des effets de bords sur des variables globales ou sur ses arguments par le caractère « `!`».

La fonction prédéfinie `cons` permet de construire une liste à l'exécution. La liste `'(1 2 3)` s'obtient donc dynamiquement par l'évaluation de l'expression `(cons 1 (cons 2 (cons 3 '()))`. Cette fonction joue un rôle particulier en Scheme puisqu'elle est la seule responsable pour l'allocation dynamique de mémoire. Elle alloue ce que l'on appelle une paire, une structure contenant deux champs : une valeur (traditionellement appelé `car`) et le champ contenant le reste de la liste après la valeur du champ `car` (traditionnellement champ `cdr`, prononcez «coudeur»).[3] Ne cherchez pas de fonction permettant de libérer l'espace en Scheme,

[3]Ces noms de champs viennent des noms des deux registres utilisés pour pointer sur ces deux éléments

elle n'existe pas. Le modèle de mémoire de Scheme prévoit qu'un glaneur de cellules (GC) récupère automatiquement l'espace alloué dynamiquement lorsqu'il devient inutile.

Il existe trois formes spéciales permettant d'introduire des liaisons de variables locales en Scheme ; elles sont introduites par `let`, `let*` et `letrec`. Toutes les trois ont la même forme générale (ici `let`) :

`(let` *Liaisons* *E*`)`

où *Liaisons* est une liste de paires identificateur–expression de la forme $(id_k\ E_k)$. Ces trois formes diffèrent dans leur sémantique. Dans la forme `let`, les expressions de définitions des variables sont toutes évaluées dans l'environnement externe au `let`, puis les variables sont liées et l'expression E est évaluée dans l'environnement externe au `let` étendue de ces nouvelles liaisons. Dans le `let*`, chaque nouvelle liaison est ajoutée à l'environnement courant avant d'évaluer l'expression de définition de la variable suivante. Dans le `letrec`, une expression de définition d'une variable peut référer récursivement à une variable définie dans le `letrec`, dans la mesure où la valeur de cette variable n'est pas nécessaire pour obtenir la valeur de l'expression (surtout utile dans les λ-expressions définie juste après).

Scheme possède de nombreuses autres formes spéciales, et quantité de fonctions prédéfinies. Le lecteur est invité à consulter le «*Revised*^{4} *Report on the Algorithmic Language Scheme*» [CR91] pour en connaître la liste complète.

2.2.2 Fonctions d'ordre supérieur

L'originalité des langages fonctionnels est leur traitement de la fonction comme entité de plein droit, c'est-à-dire une entité qu'on peut créer à l'exécution, ranger dans une variable, passer en paramètre et retourner comme résultat d'une autre fonction. Ce traitement est à la base de la puissance des langages fonctionnels.

Une fonction est créée par l'évaluation d'une forme spéciale appélée λ-expression :

`(lambda (`x_1 ... x_n`)` *E*`)`

où x_1 ... x_n sont les paramètres formels de la fonction et E son corps. Le résultat de l'évaluation d'une telle forme est une fonction *anonyme*, en ce sens qu'elle n'a pas de nom. Elle est une valeur manipulable comme les autres, ce qui permet par exemple d'écrire :

```
((if #T (lambda (x y) y) +) 3 4)
```

expression qui retourne 4. Notez en passant que l'évaluation du symbole `+` a pour résultat la fonction (anonyme !) d'addition qui, lorsqu'appliquée, retourne la somme de ses deux arguments.

Une valeur fonctionnelle peut être manipulée comme n'importe qu'elle autre valeur, et en particulier passée en paramètre ou retournée comme résultat d'une autre fonction. Une fonction qui prend une autre fonction en paramètre est dite *fonction d'ordre supérieur*. La fonction d'ordre supérieur la plus connue est la fonction `map`. Cette fonction prend deux paramètres : une fonction `f` et une liste `l`. `map` applique la fonction `f` à chacun des éléments de `l` et retourne la liste des résultats correspondants.

```
(define (map f l)
  (if (null? l)
```

de structure dans la première implantation de Lisp ; ils désignent également les deux fonctions prédéfinies permettant d'accéder respectivement au premier élément d'une liste et au reste d'une liste sans son premier élément.

```
    '()
    (cons (f (car l))
          (map f (cdr l)))))
```

Par exemple, `(map odd? '(1 2 3 4 5 6))` donne `(#T #F #T #F #T #F)`. En fait, la fonction `map` de Scheme est encore plus puissante puisqu'elle accepte une fonction `f` à n arguments et n listes en entrée et retourne la liste des résultats obtenus en appliquant la fonction `f` aux éléments correspondants des n listes. Par exemple, la fonction `zip` qui prend deux listes et retourne une liste de paires contenant les éléments correspondants des deux listes s'écrit avec `map` :

```
(define (zip l1 l2)
  (map (lambda (x1 x2) (cons x1 (cons x2 '()))) l1 l2))
```

Par exemple, `(zip '(1 2 3) '(4 5 6))` donne `((1 4) (2 5) (3 6))`. Deux autres fonctions d'ordre supérieur jouent un rôle important en programmation fonctionnelle : `fold` et `scan`. De fait, pour `fold` il faut plutôt parler de deux fonctions : `foldl` et `foldr`. La fonction `fold` prend une fonction à deux arguments, une liste et un élément de départ, et retourne le résultat de l'application (progressive) de la fonction entre tous les éléments de la liste. `foldl` prend les éléments de la liste successivement de gauche à droite alors que `foldr` les prend de droite à gauche :

```
(define (foldl f a l)
  (if (null? l)
    a
    (foldl f (f a (car l)) (cdr l))))

(define (foldr f a l)
  (if (null? l)
    a
    (f (car l) (foldr f a (cdr l)))))
```

Par exemple, `(foldl (lambda (x y) (+ x y)) 0 '(1 2 3))` donne 6, c'est-à-dire la somme de tous les éléments dans la liste. La fonction `scan` applique l'équivalent d'un `foldl` et produit comme résultat la liste de tous les résultats intermédiaires :

```
(define (scan f a l)
  (if (null? l)
    (list a)
    (cons a (scan f (f a (car l)) (cdr l)))))
```

Par exemple, `(scan (lambda (x y) (+ x y)) 0 '(1 2 3))` donne `(0 1 3 6)`.

Comme nous l'avons souligné, les fonctions d'ordre supérieur sont en grande partie à l'origine de la puissance des langages fonctionnels. Celles que nous avons vues ici concernent l'application de fonctions aux éléments d'une structure de données de type liste. Elles sont aisément modifiables pour traiter toutes sortes de structures de données récursives, dont des arborescences. Si on considère par exemple un type de données arbre de syntaxe abstraite telle que produit par la partie frontale d'un compilateur, il devient possible de décomposer un traitement sur cet arbre en deux parties : une partie parcours de l'arbre réalisée par une fonction d'ordre supérieur de type «`maparbre`» et une fonction qui s'applique à chaque noeud de cet arbre. Des compilateurs entiers sont réalisés de cette manière qui rend le traitement des noeuds indépendants de la forme précise de la structure de données arborescente. Les itérateurs maintenant bien répandus dans les langages à objets doivent beaucoup aux fonctions

d'ordre supérieur. En fait, objets et fonctions d'ordre supérieur sont deux approches différentes pour réaliser l'abstraction de données, objectif fondamental en programmation.

Fermetures

La manipulation des fonctions comme des entités de plein droit n'est pas sans conséquence sur notre conception même de ce qu'est un fonction en tant qu'entité informatique. Considérons la création de fonction à l'exécution en Scheme (sans affectation) pour mieux fixer les idées. En Scheme, la forme syntaxique `lambda` permet d'écrire des expressions de création d'abstraction. La fonction créée par l'expression :

```
(lambda (a b c) (- (* b b) (* 4 (* a c))))
```

est nulle autre que celle calculant le discriminant d'un polynôme du second degré de coefficients `a`, `b` et `c`. L'exécution de cette expression crée la fonction. Nous remarquons immédiatement que Scheme impose que le code de la fonction soit explicitement donné lors de l'écriture du programme. Il n'est pas possible d'écrire une expression `(lambda (a b c) x)` où `x` représenterait une expression forgée à l'exécution. Cette restriction permet de compiler le code des formes `lambda`, comme n'importe quel autre code ; lors de l'exécution il ne restera plus qu'à créer la fonction comme telle, ce qui va nécessiter, comme nous allons le voir immédiatement de la lier à son contexte de création.

Scheme utilise la portée lexicale des variables. Ses trois principales formes de liaison de variables, `let`, `let*` et `letrec`, utilisent cette approche à la visibilité des identificateurs. Mais dans les trois cas, il s'agit de «sucre syntaxique», puisque ces trois formes peuvent être transformées en application de fonction :

`(let ((`x_1 e_1`) ... (`x_n e_n`)) e)`

se transforme en

`((lambda (`x_1 `...` x_n`) e)` e_1 `...` e_n`)`

alors que

`(let* ((`x_1 e_1`) ... (`x_n e_n`)) e)`

se transforme en

`((lambda (`x_1`) ... ((lambda (`x_n`) e)` e_n`) ...)` e_1`)`

et finalement

`(letrec ((`x_1 e_1`) ... (`x_n e_n`)) e)`

se transforme en

`((lambda (`x_1 `...` x_n`) (set! `x_1 e_1`) ... (set! `x_n e_n`) e) '* ... '*)`

Dans chaque cas, nous en sommes donc réduits à étudier la liaison d'une variable à une valeur par le passage de paramètre lors de l'application d'une fonction. La portée lexicale impose donc que toute variable libre dans une fonction est définie par la liaison apparaissant le plus proche syntaxiquement dans les blocs lexicaux englobant. Les blocs lexicaux de Scheme sont les formes liant des variables donc les formes `lambda` ainsi que, par sucre syntaxique, `let`, `let*` et `letrec`. Dans l'extrait de programme suivant :

```
(let ((x 10) (y 20)) ((lambda (x) (+ x y)) 100))
```

La forme `let` introduit deux liaisons pour les variables `x` et `y` et forme le bloc lexical

englobant pour son corps. Ce corps est formé d'une application du résultat d'une λ-expression à la valeur 100. Cette λ-expression introduit une nouvelle liaison pour la variable `x` qui, par le passage de paramètre est liée à 100. La variable `y` du corps de la fonction est libre dans la λ-expression ; la liaison utilisée pour trouver sa valeur est donc celle introduite par le bloc lexical englobant, donc le `let` liant `y` à 20. Le résultat de l'expression est finalement 120.

De cet exemple, il apparaît clairement qu'une fonction doit être exécutée dans l'environnement de définition de cette fonction. Comment retrouver cet environnement ? L'idée est de capturer dans la représentation de la fonction son environnement de définition. Cette représentation de la fonction est justement appelée *fermeture*, car les variables libres de la fonction sont liées dans l'environnement de définition ; la fermeture est donc complète en ce qui concerne les liaisons de variables.

2.2.3 Style passage de la continuation

Considérons la fonction factorielle récursive bien connue :

```
(define fact
  (lambda (n)
    (if (zero? n)
      1
      (* n (fact (- n 1))))))
```

Il est possible d'utiliser une dérivation pour décrire le calcul réaliser par `fact`, par exemple sur le paramètre réel `4` :

```
    (fact 4)
⇒   (* 4 (fact 3))
⇒   (* 4 (* 3 (fact 2)))
⇒   (* 4 (* 3 (* 2 (fact 1))))
⇒   (* 4 (* 3 (* 2 (* 1 (fact 0)))))
⇒   (* 4 (* 3 (* 2 (* 1 1))))
⇒   (* 4 (* 3 (* 2 1)))
⇒   (* 4 (* 3 2))
⇒   (* 4 6)
⇒   24
```

Chaque appel à la fonction `fact` se fait dans un *contexte*. Si on considère le calcul de `(fact 0)`, il se fait dans le contexte `(* 4 (* 3 (* 2 (* 1 □))))` où □ représente l'expression `(fact 0)` ou plus précisément la valeur attendue par le contexte comme résultat de l'expression `(fact 0)`. Un contexte se représente facilement sous la forme d'une λ-expression :

```
(lambda (v) (* 4 (* 3 (* 2 (* 1 v)))))
```

où le paramètre `v` représente la valeur à insérer dans le contexte et où l'application de la fonction ainsi créée revient à relancer l'exécution du contexte. Soit `k` cette λ-expression, elle permet de traduire l'expression `(* 4 (* 3 (* 2 (* 1 (fact 0)))))` en `(k (fact 0))`. De fait, si nous généralisons, chacune des lignes ci-haut contenant un appel à `fact` sous la forme `(`k `(fact` n`))` avec les valeurs de k et n appropriée. Cette forme nous permet d'introduire une nouvelle définition de `fact` en style *passage à la continuation* (ou CPS pour *continuation-passing style*). Le style passage à la continuation est basé sur l'introduction de la continuation comme un paramètre explicite de chaque fonction, sur le fait de passer le résultat de la fonction directement à la continuation lorsque ce résultat est obtenue par exécution d'une expression et

sur le fait de construire une nouvelle continuation à chaque fois qu'on appelle une fonction pour poursuivre le calcul. La fonction `fact-cps` prend donc un paramètre `k` pour la continuation explicite et gère explicitement cette continuation dans son corps :

```
(define fact-cps
  (lambda (n k)
    (if (zero? n)
      (k 1)
      (fact-cps (- n 1) (lambda (v) (k (* n v)))))))
```

Il est intéressant de noter que la transformation d'un programme en style direct (normal) en style passage à la continuation est automatisable. Mais qu'est-ce donc qu'une continuation ? Est-ce une simple fonction ?

Pour répondre à cette question, considérons une nouvelle forme de procédure appelée *procédure d'échappement*. Une procédure d'échappement est simplement une procédure dont le résultat, lorsqu'elle est appelée devient le résultat de l'ensemble du calcul en cours. Soit par exemple une procédure d'échappement `echappe-*` qui réalise la multiplication de ses arguments. Le résultat de l'expression :

```
(+ 2 (* 3 (echappe-* 4 5)))
```

est donc 20 puisque la multiplication de 4 et 5 donne 20 et, puisque `echappe-*` est une procédure d'échappement, le reste du calcul est tout simplement ignoré.

Par définition, une continuation est un contexte représenté par une procédure d'échappement. Plus généralement, en tout point de programme, la continuation représente le reste de l'exécution du programme en ce point. Dans les implantations que nous avons vus jusqu'à maintenant, la continuation est représentée par la pile d'éxécution. En effet, examinons l'accumulation des continuations intermédiaires dans le calcul de `(fact-cps 3 ik)` où `ik` est la continuation identité représentée par a procédure d'échappement `(lambda (v) v)`. Par dérivation, on obtient :

```
    (fact-cps 3 (λv. v))
⇒   (fact-cps 2 (λv. ((λv. v) (* 3 v))))
⇒   (fact-cps 1 (λv. ((λv. ((λv. v) (* 3 v))) (* 2 v))))
⇒   (fact-cps 0 (λv. ((λv. ((λv. ((λv. v) (* 3 v))) (* 2 v))) (* 1 v))))
⇒   ((λv. ((λv. ((λv. ((λv. v) (* 3 v))) (* 2 v))) (* 1 v))) 1)
⇒   ((λv. ((λv. ((λv. v) (* 3 v))) (* 2 v))) (* 1 1))
⇒   ((λv. ((λv. ((λv. v) (* 3 v))) (* 2 v))) 1)
⇒   ((λv. ((λv. v) (* 3 v))) (* 2 1))
⇒   ((λv. ((λv. v) (* 3 v))) 2)
⇒   ((λv. v) (* 3 2))
⇒   ((λv. v) 6)
⇒   6
```

L'imbrication des continuations successives fait clairement apparaître la discipline de pile. Le fait est que les continuations peuvent être représentées par des procédures d'échappement ou comme des structures de données. Dans ce second cas, elles sont représentées par un enchainement de blocs d'activations attendant un résultat et représentant le point de programme où le calcul doit reprendre dans la fonction avec le résultat reçu de l'appelé.

Le principal intérêt du style de programme en passage à la continuation est le fait qu'il rende explicite le séquencement de tous les calculs. Chaque opération apparait alors explicitement avec son contexte d'exécution qui attend son résultat avant de poursuivre le calcul avec

la prochaine opérations. Par exemple, les langages de programmation laissent la plupart du temps non-spécifié l'ordre d'évaluation des paramètres réels lors d'un appel de fonction. En style passage à la continuation, l'ordre d'evaluation doit apparaître explicitement. Ce facteur est d'ailleurs la raison pour laquelle la forme CPS est parfois utilisée comme forme intermédiaire faisant apparaître l'ensemble du flôt de contrôle explicitement. Par contre, toute cette information devenue explicite rend les programmes très lourds, ce qui limite une utilisation de ce style dans les programmes écrits par les humains (et non générés automatique par une transformation comme dans le cas de la forme intermédiaire dans les compilateurs).

L'utilisation du concept de continuation ne se limite pas au style passage à la continuation. Plutôt que de les faire apparaître ainsi, un langage peut fournir des primitives permettant de capturer la continuation courante en un point de programme et de les invoquer par la suite. Le principal intérêt d'une telle manipulation des continuations réside dans la capacité qu'elle donne d'exprimer toutes sortes de structure de contrôle. En particulier, elles ont été utilisées pour modéliser les structures de contrôle courantes comme l'alternative et l'itération mais aussi des structures de contrôle quasi-parallèle comme l'échappement lors d'erreur ou les coroutines. D'un point de vue puissance d'expression mais aussi de difficulté d'implantation, on distingue deux types de continuations : les continuations à durée de vie dynamique et les continuations à durée de vie illimitée.

Les continuations à durée de vie dynamique ne sont invocables que si elles sont encore présentes dans la pile, c'est-à-dire qu'elles ne sont invocables que dans la fonction qui les a capturées ou dans une fonction appelée par celle-ci. Les continuations à durée de vie illimitée peuvent être invoquées à tout moment durant l'exécution d'un programme. Si on voit une continuation comme une pile d'exécution, la durée de vie illimitée force l'implantation à conserver une copie de toute la pile capturée, qui peut être réinstaurée à tout moment. Elles sont donc beaucoup plus difficiles à implanter efficacement que les continuations à durée de vie dynamique qui n'exige pas ce type de copie.

Scheme propose une fonction prédéfinie appelée `call-with-current-continuation`, souvent abrégée en `call/cc`, qui permet de capturer la continuation courante à durée de vie illimitée. Outre les fonctions, l'habileté de Scheme à manipuler la continuation également comme une entité de plein droit en fait un langage d'une puissance sans aucune commune mesure avec l'économie de concept dont il fait preuve. Scheme symbolise une tendance particulière en conception de langages de programmation qui vise petit mais puissant.

2.3 Meta-interprète pour Scheme

Scheme est un lmangage puissant et concis. Il se caractérise par une économie des concepts tout en préservant une très large spectre dans leur complexité. Retenons par exemple les fonctions et les continuations comme entité de plein droit. Sa mise en oeuvre a donc fait l'objet de nombreuses études où sa concision est mise à profit pour explorer de nouvelles approches d'implantation de ses puissants concepts.

Dans cette section, nous allons brièvement présenter un petit interprète d'un sous-ensemble fonctionnel pur de Scheme. Cet interprète a la particularité d'être méta-circulaire, ce qui veut simplement dire qu'il est capable de s'interpréter lui-même (après tout, il est aussi un programme Scheme!). Nous ne pourrons qu'effleurer ici le sujet de la mise en oeuvre de Scheme. Friedman, Wand et Haynes [FWH92] proposent une introduction à la mise en oeuvre des langages à la portée de tout étudiant de second cycle en informatique. Queinnec [Que94] a produit une remarquable bible sur la mise en oeuvre non seulement de Scheme mais aussi

d'une couche objet pour Scheme ; tout étudiant de troisième cycle intéressé par la mise en oeuvre des langages de programmation (impératifs, fonctionnels ou à objets) se doit d'avoir assimilé ce livre.

L'écriture d'un méta-interprète, ou plus généralement d'un interprète, pour un langage est une des méthodes bien connues pour définir une sémantique opérationnelle. Cette approche, très en vogue dans les années '70 et '80, est maintenant supplantée, dans la mesure du possible, par des approches plus formelles. Elle va néanmoins nous servir d'introduction à la mise en oeuvre des concepts les plus simples et la similarité de Scheme et du λ-calcul va préparer notre passage à la sémantique dénotationnelle.

La structure d'un interprète peut varier, mais l'approche la plus courante consiste à coller à la structure syntaxique du langage. On dit alors que la structure de l'interprète est dirigée par la syntaxe, ce qui est aussi une caractéristique de la plupart des approches formelles. Notre sous-ensemble de Scheme ne reconnaît que neuf formes syntaxiques : deux expressions de base (constantes et variables), l'appel de fonction (ou expression standard), et six formes spéciales (`quote`, `if`, `cond`, `let`, `let*`, et `lambda`).

Le coeur de l'interprète est donc constitué de la fonction d'aiguillage `val` dont le rôle consiste à identifier la forme syntaxique de l'expression courante et, cette identification faite, d'appeler une fonction appropriée à l'expression en question. La fonction `val` prend deux paramètres : une expression courante et un environnement de liaison des variables.

```
(define val
  (lambda (exp env)
    (cond ((constant? exp)    (val-constant exp env))
          ((quote? exp)       (val-quote exp env))
          ((variable? exp)    (val-variable exp env))
          ((if? exp)          (val-if exp env))
          ((cond? exp)        (val-cond exp env))
          ((let? exp)         (val-let exp env))
          ((let*? exp)        (val-let* exp env))
          ((lambda? exp)      (val-lambda exp env))
          ((application? exp) (val-application exp env))
          (else (error "expression inconnue:" exp)))))
```

Nos programmes sont représentés par des listes Scheme. Nous allons donc nous servir des fonctions prédéfinies de Scheme pour identifier les constantes (ici uniquement les nombres) et les variables représentées par des symboles.

```
(define constant? number?)

(define variable? symbol?)
```

Les formes spéciales se présentent comme des listes possèdant toutes un identificateur particulier en position d'opérateur. Les prédicats chargés de les identifier peuvent donc se contenter de comparer le symbole en première position dans la liste. Nous introduisons donc une fonction `starts-with?` comparant un symbole avec le premier élément d'une liste, que nous utilisons ensuite dans les prédicats d'identification des formes spéciales :

```
(define starts-with?
  (lambda (exp symbol)
    (and (pair? exp) (equal? symbol (car exp)))))
```

```
(define quote? (lambda (exp) (starts-with? exp 'quote)))

(define if? (lambda (exp) (starts-with? exp 'if)))

(define cond? (lambda (exp) (starts-with? exp 'cond)))

(define let? (lambda (exp) (starts-with? exp 'let)))

(define let*? (lambda (exp) (starts-with? exp 'let*)))

(define lambda? (lambda (exp) (starts-with? exp 'lambda)))
```

L'application est alors une liste (une paire) ne débutant pas par l'un des identificateurs précédents. Si l'on tient compte de l'ordre dans lequel on applique les prédicats, en plaçant le test pour l'application à la fin, il suffit de vérifier que l'expression est bien une liste.

```
(define application? pair?)
```

Une forme interne à l'interprète est introduite pour représenter les fermetures. Ces formes se composent du symbole `closure` en première position suivi de la λ-expression et de l'environnement de définition de cette fonction.

```
(define closure? (lambda (exp) (starts-with? exp 'closure)))
```

Muni de ces prédicats nous permettant d'identifier les différentes formes syntaxiques dans notre représentation sous forme de liste, nous pouvons maintenant passer aux fonctions d'évaluations de ces différentes formes.

Les formes constantes et citation sont les plus simples. La valeur d'une constante est cette constante même, alors que la valeur d'une forme citation est simplement la forme citée.

```
(define val-constant
  (lambda (exp env)
    exp))

(define val-quote
  (lambda (exp env)
    (cadr exp)))
```

Pour définir l'évaluation d'une forme variable, il nous faut en dire un peu plus sur la représentation de notre environnement. Nous choisissons d'utiliser les listes d'associations prédéfinies en Scheme, c'est-à-dire une liste de paires identificateur–valeur. L'avantage de cette représentation est que nous disposons de fonctions prédéfinies (`assoc`) pour les traiter.

Nous définissons donc deux fonctions : `bind` créant une liaison identificateur–valeur et `bind-all` capable de lier, un à un, une liste d'identificateurs et une liste de valeurs (en positions correspondantes) :

```
(define bind
  (lambda (var value env)
    (cons (list var value) env)))

(define bind-all
  (lambda (vars values env)
    (append (map list vars values) env)))
```

On notera dans les deux cas que les nouvelles liaisons sont ajoutées au début de la liste d'associations, ce qui nous permettra d'étendre l'environnement par masquage des liaisons précédentes, comme on l'attend d'un langage à portée lexicale.

La valeur d'une forme variable est donc la plus récente liaison de cette variable dans la liste d'association servant alors d'environnement courant.

```
(define val-variable
  (lambda (exp env)
    (let ((found (assoc exp env)))
      (if (pair? found)
        (cadr found)
        (error "unknown variable:" exp)))))
```

La valeur d'une forme conditionnelle est directement réalisée à l'aide de la conditionnelle de Scheme :

```
(define val-if
  (lambda (exp env)
    (if (val (cadr exp) env)
       (val (caddr exp) env)
       (val (cadddr exp) env))))
```

Les puristes noteront ici, comme en d'autres endroits de cet interprète, que l'ordre d'évaluation du langage de définition devient alors l'ordre d'évaluation du langage défini. Une façon d'éviter ce problème serait de réécrire tout l'interprète en style passage de la continuation.[4]

La forme `cond` est traitée en deux temps. D'abord, la fonction `val-cond` extrait la liste des paires conditions-expressions de la forme `cond`, puis elle appelle la fonction `val-cond-r` qui sera chargée de traiter une à une ces paires :

```
(define val-cond
  (lambda (exp env)
    (val-cond-r (cadr exp) env)))
```

La fonction `val-cond-r` explore récursivement la liste des paires une à une. Elle extrait la condition et le conséquent, puis teste si la condition est le mot-clé `else` ; si oui, elle exécute le conséquent dont le résultat devient le résultat de la forme. Si la condition n'est pas `else`, il faut évaluer la condition, et selon qu'elle est vraie ou fausse, on exécute le conséquent courant ou on passe au reste de la liste des paires conditions-expressions :

```
(define val-cond-r
  (lambda (exps env)
    (let* ((exp (car exps))
           (condition (car exp))
           (consequent (cadr exp)))
      (if (equal? condition 'else)
        (val consequent env)
        (if (val condition env)
          (val consequent env)
          (val-cond-r (cdr exps) env))))))
```

Le traitement de la forme `let` réalisé par la fonction `val-let` consiste à récupérer la liste

[4] Pourquoi ?

des variables définies par la forme, puis la liste des expression associées. Les expression sont évaluées dans l'environnement de départ pour donner la liste de valeurs `values`. Un nouvel environnement `new-env` est alors obtenu en liant les variables à ces valeurs ; ce nouvel environnement sert enfin à évaluer l'expression formant le corps du `let` :

```
(define val-let
  (lambda (exp env)
    (let* ((vars (map car (cadr exp)))
           (exprs (map cadr (cadr exp)))
           (values (map (lambda (e) (val e env)) exprs))
           (new-env (bind-all vars values env)))
      (val (caddr exp) new-env))))
```

Pour la forme `let*`, le traitement est légèrement différent. La fonction `val-let*` défère à la fonction `val-let*-env` le soin de calculer l'environnement obtenu à partir des liaisons et se contente alors d'évaluer le corps de la forme `let*` dans ce nouvel environnement :

```
(define val-let*
  (lambda (exp env)
    (val (caddr exp) (val-let*-env (cadr exp) env))))
```

La fonction `val-let*-env` procède de façon récursive en traitant la liste des liaisons. Pour chaque liaison, elle évalue d'abord l'expression dans l'environnement calculé à partir de l'environnement extérieur à la forme étendu de toutes les liaisons traitées depuis le début de la liste. Lorsque la liste de liaisons est épuisée, l'environnement obtenu contient toutes les liaisons des variables de la forme :

```
(define val-let*-env
  (lambda (pairs env)
    (if (null? pairs)
      env
      (let ((new-env (bind (caar pairs) (val (cadar pairs) env) env)))
        (val-let*-r (cdr pairs) new-env)))))
```

L'évaluation de la forme `lambda` est relativement simple. Il s'agit de créer une fermeture, c'est-à-dire une liste débutant par le symbole `'closure` et qui contiendra la λ-expression et l'environnement courant, soit l'environnement de définition de la fonction :

```
(define val-lambda
  (lambda (exp env)
    (list 'closure exp env)))
```

L'application est plus complexe. Une application demande d'abord à évaluer l'expression en position fonctionnelle, ce qui doit retourner une fermeture ou une procédure primitive à appliquer, puis les paramètres réels de l'appel (`actuals`). Deux types de fonctions sont prévues par notre méta-interprète : les procédures primitives (connues dès le départ) et les fermetures (créées par les formes `lambda`) :

```
(define val-application
  (lambda (exp env)
    (let ((op (val (car exp) env))
          (actuals (map (lambda (x) (val x env)) (cdr exp))))
      (cond ((procedure? op) (apply op actuals))
            ((closure? op) (apply-lambda op actuals))
```

```
        (else (error "val-application: on" exp))))))
```

Les procédures primitives sont appliquées par une fonction auxiliaire `apply`, alors que les fermetures sont appliquées par la fonction `apply-lambda`. La fonction `apply-lambda` reçoit la fermeture et les paramètres réels évalués. Elle récupère l'environnement de définition de la lambda, qu'elle étend par les liaisons des paramètres formels aux paramètres réels (passage par valeur) pour ensuite évaluer le corps de la lambda dans ce nouvel environnement :

```
(define apply-lambda
  (lambda (clos actuals)
    (let* ((lam (cadr clos))
           (def-env (caddr clos))
           (formals (cadr lam))
           (body (caddr lam))
           (new-env (bind-all formals actuals def-env)))
      (val body new-env))))
```

Exercices

1. Introduire la notion de bloc d'activation dans le méta-interprète Scheme en voyant les valeurs des arguments d'une fonction comme un vecteur et les références à ces arguments comme un déplacement dans ce vecteur.
2. Introduire la forme spéciale «`set !`» au méta-interprète.
3. Transformer le méta-interprète en style passage de la continuation (CPS).

Chapitre 3

Le λ-calcul

Notre objectif est de munir des langages de programmation d'une sémantique formelle. Pour cela, il faut exprimer cette sémantique à l'aide d'une certaine notation. Pourtant, s'en remettre à une tierce notation fait en sorte que la définition n'est pas plus formelle que cette dernière. Dans l'approche opérationnelle que nous avons introduite au chapitre 2 avec notre méta-interprète pour Scheme, la notation utilisée pour définir la sémantique de Scheme est assez singulière puisque c'est Scheme lui-même ! Cette auto-définition est souvent utilisée en informatique, et elle se prête à plusieurs applications, dont un test de puissance expressive du langage et la construction d'interprètes ou de compilateurs par la technique d'auto-amorçage.

La sémantique dénotationnelle utilise plutôt le λ-calcul comme notation pour écrire les fonctions et les domaines, sur lesquels nous reviendrons plus loin, comme «structures de données». Pourquoi le λ-calcul nous fournit-il une base plus solide que Scheme, Lisp, ou un autre langage ? En fait, le λ-calcul est lui-même fondée sur une sémantique formelle. L'utilisation de cette notation procède donc d'un stratagème bien connu en mathématique (et en informatique !) qui consiste à construire une théorie par couches successives, passant de concepts fondamentaux qui permettent de dériver formellement des concepts plus complexes, et ainsi de suite jusqu'à l'obtention de la théorie désirée.

3.1 Fondements

En théorie des ensembles, une fonction est définie par son graphe qui exhibe son comportement entrée/sortie. Par exemple, une fonction unaire est définie par un ensemble de paires, où chaque paire contient une valeur d'argument dans sa première composante et le résultat correspondant dans sa seconde composante. On peut généraliser ce schéma à des fonctions à n arguments en utilisant un n-tuple de valeurs pour représenter les arguments de la fonction. Par exemple, la fonction d'addition sur les entiers naturels se défini par :

$$\{((0,0),0),((0,1),1),...,((1,1),2),...\}$$

Ce genre de définitions dites extensionnelles, mène naturellement à une théorie des fonctions qui inclut une notion d'égalité extensionnelle selon laquelle deux fonctions sont égales si et seulement si elles ont le même graphe.

L'approche extensionnelle, quoique satisfaisante pour raisonner théoriquement sur les fonctions, est de peu d'intérêt lorsqu'il s'agit d'aborder le calcul des fonctions, que ce soit théo-

riquement en calculabilité, ou pratiquement en informatique. Pour aborder ces notions, il faut pouvoir définir une fonction en terme d'un algorithme ou d'une règle décrivant comment calculer un résultat à partir des arguments. Deux fonctions sont alors égales si elles sont définies par les mêmes règles (ou des règles équivalentes), ce qui introduit une notion d'égalité intensionnelle.

Plusieurs formalismes ont été proposés pour définir les fonctions en termes de règles de calcul ; le λ-calcul est l'un de ceux-là, mais un autre formalisme bien connu en informatique est la machine de Turing. Le λ-calcul est plus précisément un système formel où une fonction est exprimée en termes de régles de correspondance entre arguments et résultat. Un système formel se décline selon trois principaux aspects :

Notation : une syntaxe permettant de contruire les termes qui, dans le cas du λ-calcul représentent les régles de calcul. Elle consiste en deux parties : les symboles (alphabet) et la syntaxe des termes eux-mêmes, définie par une grammaire.

Théorie : un ensemble d'axiomes et de règles permettant de relier les termes entre eux ce qui, dans le cas du λ-calcul, permet de raisonner sur une forme d'équivalence entre les termes liée à une notion de calcul. Une théorie comprend un certain nombre de théorèmes donnés, appelés axiomes, et un ensemble de règles d'inférence qui permettent de dériver de nouveaux théorèmes à partir des théorèmes préalablement obtenus.

Modèles : une sémantique mathématique dont on va munir le système. Le but d'un modèle est de donner une signification aux termes. Une interprétation est utilisée pour définir la valeur que chaque terme dénote. Si tous les théorèmes d'une théorie sont valides selon l'interprétation, alors celle-ci fournit un modèle pour la théorie.

Outre le λ-calcul, des systèmes formels familiers sont le calcul propositionnel et le calcul des prédicats du premier ordre en logique mathématique. Si les notions de notation et de théorie sont généralement bien connues, celle de modèle peut être moins familière. Un exemple d'interprétation en logique propositionnelle est l'attribution d'une valeur de vérité, vrai ou faux, à chaque proposition ; une interprétation est alors un modèle si toutes les propositions valides selon la théorie ont pour valeur de vérité vrai dans l'interprétation.

Pour le λ-calcul, le notion de modèle est plus complexe et fait appel à des dénotations qui ne sont pas de simples ensembles. En fait, l'une des approches pour construire un modèle du λ-calcul consiste à utiliser la théorie des domaines, aussi utilisée en sémantique dénotationnelle. Nous reviendrons donc sur ce modèle du λ-calcul plus loin.

3.2 Syntaxe des λ-termes

L'ensemble Λ des termes du λ-calcul, aussi appelés λ-termes, est constitué de mots formés à partir de l'alphabet suivant :

variables	$x, y, z, ...$
λ	
$'$	
.	
$(,)$	parenthèses

La syntaxe des termes du λ-calcul est donnée par la grammaire suivante :

$$\begin{array}{rll} terme & ::= variable & \\ & \mid \lambda\, variable\,.\, terme & \textbf{abstraction} \\ & \mid (terme\ terme) & \textbf{application} \\ variable & ::= x \mid y \mid z... & \end{array}$$

La seconde forme des termes est appelée abstraction car elle permet de créer des fonctions, alors que la troisième est appelée application car elle permet d'appliquer une fonction, représentée par le premier terme entre parenthèses, à un argument. Par exemple, x, $(x\,y)$ et $\lambda x.\,x$ sont des λ-termes.

La grammaire précédente engendre un ensemble de termes que l'on peut aussi définir inductivement par :

Définition 3.1 *(λ-termes)*

L'ensemble des λ-termes est le plus petit ensemble Λ tel que :

1. $x \in \Lambda$, *pour x variable*
2. *si* $M \in \Lambda$, *alors* $\lambda x.M \in \Lambda$
3. *si* $M, N \in \Lambda$, *alors* $(M\ N) \in \Lambda$

Cette définition récursive permet de construire l'ensemble Λ par un calcul de point fixe qui consiste à initialiser Λ à l'ensemble des variables, puis à appliquer répétitivement les règles d'abstraction et d'application pour construire de nouveaux termes à partir des termes précédents. À la limite, ce processus convergence vers l'ensemble cherché.

Le fait que la syntaxe ci-dessus limite les fonctions à n'avoir qu'un seul argument n'est limitatif qu'en apparence. En fait, on doit à Schönfinkel l'idée popularisée ensuite par Curry qu'une fonction à plusieurs arguments peut s'écrire en une succession de fonctions en un argument. Par exemple, une fonction à deux arguments pour additionner deux nombres peut s'écrire : $\lambda x\,.(\lambda y\,.\,((+\,x)\,y)$. On dit alors que la fonction à plusieurs arguments est écrite sous forme *currifiée*, une forme qui revient à prendre un argument à la fois.

Le symbole λ joue ici un rôle similaire aux symboles $\exists$ et $\forall$ du calcul des prédicats du premier ordre, ou encore à la notation $\int ...\,dx$ en calcul intégral, c'est-à-dire qu'il agit pour délimiter la portée d'une variable dans un terme. Une variable bornée par le symbole λ dans un terme est dite *liée*. On définit inductivement l'ensemble des variables liées dans un terme par la fonction $VB : \Lambda \to \mathcal{P}(Variable)$[1] :

$$\begin{array}{rcl} VB(x) & = & \varnothing \\ VB(\lambda\, x\,.\, M) & = & VB(M) \cup \{x\} \\ VB(MN) & = & VB(M) \cup VB(N) \end{array}$$

De manière similaire, on définit l'ensemble des variables non-liées, ou *libres*, par la fonction $VL : \Lambda \to \mathcal{P}(Variables)$:

$$\begin{array}{rcl} VL(x) & = & \{x\} \\ VL(\lambda\, x\,.\, M) & = & VL(M) - \{x\} \\ VL(M\ N) & = & VL(M) \cup VL(N) \end{array}$$

[1] Pour un ensemble E, $\mathcal{P}(E)$ est l'ensemble des sous-ensembles de E.

Un terme qui ne posséde pas de variables libres est appelé *combinateur* et l'ensemble de tous les combinateurs est noté Λ^0.

Syntaxiquement, une dernière entité s'avère utile dans le traitement subséquent de λ-termes : les *contextes*. Un contexte est essentiellement un terme avec un «trou» qui pourra être rempli avec un (autre) terme :

Définition 3.2 ***(contextes)***

L'ensemble des contextes est le plus petit ensemble $\mathcal{C}[]$ tel que :

1. $x \in \mathcal{C}[]$, *pour toute variable* x
2. $[] \in \mathcal{C}[]$
3. *si* $\mathcal{C}_1[], \mathcal{C}_2[] \in \mathcal{C}[]$ *alors* $(\mathcal{C}_1[]\ \mathcal{C}_2[])$, $\lambda x.\mathcal{C}_1[] \in \mathcal{C}[]$.

Le «remplissage» du «trou» dans un contexte $\mathcal{C}[]$ est le remplacement systématique du trou $[]$ par un terme t, ce qui est noté $\mathcal{C}[t]$. Par exemple, soit le contexte $c = \lambda x.[]x$, alors $c[\lambda y.y] = \lambda x.(\lambda y.y)x$. Il est important de noter que si un terme M a des variables libres, ces variables peuvent devenir liées dans un certain contexte. On dit alors que ces variables ont été *capturées*.

L'opération de *substitution* consiste à remplacer une variable libre $x \in VL(M)$ dans un terme M par un autre terme N ; elle est notée $M[x := N]$. Cette opération est analogue au «remplissage» d'un trou dans un contexte et soulève le même problème de capture potentielle des variables. Si le terme N contient des variables libres, il convient d'éviter leur capture lors de la substitution.

Plusieurs approches ont été proposées pour circonscrire la capture lors des substitutions. L'approche classique de Church fait appel à un remplacement des variables liées lorsqu'elles entrent en conflit avec des variables libres du terme à substituer. Par exemple, dans la substitution $(\lambda y.\,(y\ x))[x := y]$, la variable y que l'on veut substituer à x va être capturée par le paramètre y de la fonction interne. Mais dans le terme $\lambda y.\,(y\ x)$, le nom de variable y est immatériel. Nous aurions tout aussi bien pu écrire $\lambda z.\,(z\ x)$ sans changer la signification du terme. Ceci nous mène à la notion de changement de variables liées et d'α-congruence :

Définition 3.3 ***(changement de variable liée)***

M' est obtenu de M par un changement de variable liée si $M = \mathcal{C}[\lambda x.N]$ et $M' = \mathcal{C}[\lambda y.N[x := y]]$ où $\mathcal{C}[]$ est un contexte à un trou et y n'apparaît nulle par dans N.

Définition 3.4 ***(α-congruence)***

Deux termes M et N sont α-congruent, noté $M \equiv_\alpha N$ si N résulte de M par une série de changements de variables liées.

L'α-congruence nous permet de modifier à volonté le nom d'une variable liée, si on procède à un renommage systématique, c'est-à-dire le changement de variable liée. En utilisant cette possibilité, la substitution sans capture de variables peut se définir par :

1. $x[x := N] \equiv N$
2. $x[y := N] \equiv x$, si $x \neq y$
3. $(\lambda x.M)[x := N] \equiv \lambda x.M$
4. $(\lambda x.M)[y := N] \equiv \lambda x.M[y := N]$ si $y \notin VL(M)$ ou $x \notin VL(N)$
5. $(\lambda x.M)[y := N] \equiv \lambda z.(M[x := z])[y := N]$ si $y \in VL(M)$ et $x \in VL(N)$, où z est une nouvelle variable
6. $(M_1\ M_2)[x := N] \equiv (M_1[x := N]\ M_2[x := N])$

$$(\lambda x.M)N = M[x := N] \qquad (\beta)$$

$$M = M$$

$$\frac{M = N}{N = M}$$

$$\frac{M = N \qquad N = L}{M = L}$$

$$\frac{M = N}{(M\ Z) = (N\ Z)}$$

$$\frac{M = N}{(Z\ M) = (Z\ N)}$$

$$\frac{M = N}{\lambda x.M = \lambda x.N} \qquad (\xi)$$

FIG. 3.1 – Axiomes de la théorie λ de l'égalité sur les λ-termes.

3.3 Théorie λ

Muni de ces notions syntaxiques, une théorie de l'égalité sur les termes peut être développée de telle façon qu'un terme de type application soit égal à l'application de la fonction à ses arguments, que l'égalité soit une relation d'équivalence et que des termes égaux le soient dans n'importe quel contexte. Une telle théorie λ est proposée à la figure 3.1. La règle ξ est parfois appelée la règle d'extensionalité faible, alors que la règle β correspond à l'application de fonction. Lorsque deux termes M et N peuvent être démontrés égaux en utilisant les règles de la théorie λ, on écrit :

$$\lambda \vdash M = N$$

On dit que $M = N$ est un théorème déductible de la théorie λ.

La théorie λ permet d'établir une relation d'égalité de termes qui fait appel en quelque sorte à une notion de «convertibilité». Un terme M est égal à un terme N selon la théorie λ s'il est possible de «convertir» M en N en appliquant 0, 1, ou plusieurs pas d'inférence. Un pas d'inférence consiste à appliquer l'une des règles de la théorie sur un terme courant pour le transformer en un nouveau terme. Par exemple, les termes $(\lambda x.\, x)y$ et y sont égaux. L'application de la règle β sur le terme $(\lambda x.\, x)y$ donne $x[x := y]$ ce qui par définition se transforme en y. Notons la différence entre équivalence syntaxique et égalité selon la théorie λ. Deux termes équivalents sont égaux, mais deux termes égaux ne sont pas nécessairement équivalents :

$$M \equiv N \Rightarrow M = N$$

$$\neg(M = N \Rightarrow M \equiv N)$$

Les termes de notre exemple précédent illustre bien la seconde assertion : $(\lambda x.\, x)y = y$ mais $\neg((\lambda x.\, x)y \equiv y$, c'est-à-dire que les deux termes sont égaux mais il ne sont pas identiques.

Égalité extensionnelle

La théorie λ cherche à formaliser une relation d'égalité qui dans un certain sens rend égaux deux termes qui encode le même algorithme. Cette théorie ne permet cependant pas de déduire l'égalité de certains termes qui paraissent pourtant égaux. Considérons par exemple le terme :

$$\lambda x.Mx$$

où $x \notin VL(M)$, c'est-à-dire que M ne contient pas d'occurence de la variable liée x. Les termes $\lambda x.Mx$ et M devraient être égaux puisque si on les applique au terme N, dans les deux cas on obtient le terme MN. C'est la notion classique d'égalité extensionnelle qui dit que deux fonctions dans la théorie des ensembles sont égales si elles possèdent le même graphe. Pourtant, la formule $\lambda x.Mx = M$ n'est pas un théorème de λ.

Deux approches ont été proposées pour intégrer la notion d'égalité extensionnelle à la théorie λ. La première consiste à ajouter une nouvelle règle à la théorie pour donner une nouvelle théorie appelée $\lambda + ext$:

$$\frac{Mx = Nx}{M = N} \qquad x \notin VL(MN) \quad (ext)$$

La seconde approche consiste à ajouter un autre axiome, ce qui donne une autre théorie appelée $\lambda\eta$:

$$\lambda x.Mx = M,\ x \notin VL(M) \quad (\eta)$$

En réalité, ces deux théories sont équivalentes :

Théorème 3.1 *$\lambda + ext$ et $\lambda\eta$ sont équivalents.*

Preuve. *$\lambda + ext \vdash \lambda x.Mx = M,\ x \notin VL(M)$ car $(\lambda x.Mx)x = Mx$ par β et si $x \notin VL(M)$ alors $\lambda x.Mx = M$ par ext.*

$\lambda\eta \vdash ext$ car si on suppose que $Mx = Nx$ avec $x \notin VL(MN)$ alors $\lambda x.Mx = \lambda x.Nx$ par ξ et alors en appliquant η deux fois on obtient $M = N$.

Consistence et complétude

Avec la syntaxe des λ-termes, on peut construire l'ensemble de toutes les formules possibles selon la théorie d'égalité. Pour qu'une théorie ait une quelconque utilité, il doit y avoir au moins une formule qui soit un théorème mais toute formule ne doit pas être un théorème. Ces propriétés se retrouvent sous la notion de consistence de la théorie. La théorie λ, de même que la théorie $\lambda\eta$ ont toutes deux été démontrées consistantes (il existe des théorèmes mais toute paire de termes ne sont pas nécessairement égaux selon ces théories). Les deux ont également été démontrées complètes (si deux termes M et N ont chacun une forme normale alors soit $\lambda \vdash M = N$ ou encore $\lambda + M = N$ est une théorie qui mène à une contradiction).

3.4 Réduction

Les théories proposées précédemment permettent de raisonner sur l'égalité des λ-termes mais ne permettent pas directement de faire des calculs au sens informatique du terme. Un

axiome joue pourtant un rôle central dans la théorie λ : l'axiome β. L'essentiel du calcul en λ-calcul consiste en application de fonctions. L'axiome β indique quel terme peut être démontré égal à une application de fonction, et peut donc servir de base au calcul en indiquant comment éliminer les sous-termes applications d'un terme représentant l'ensemble du calcul à faire. Un terme dans lequel il ne reste plus d'applications de fonctions sera alors considéré comme une valeur terminale, résultat du calcul, appelée forme normale.

Définition 3.5 ***(forme normale)***

Si $M \in \Lambda$, alors M est une β-forme normale, notée β-fn, si M n'a pas de sous-terme de la forme $(\lambda x.R)\, S$.

Si $M \in \Lambda$, alors M possède une β-forme normale s'il existe un terme N tel que $M = N$ et N est une β-forme normale.

D'un point de vue pratique, une forme normale est un terme dans lequel il ne reste plus d'applications de fonctions à réaliser. Un tel terme peut être considéré comme une valeur finale puisqu'aucune règle ne permet de trouver un terme structurellement «plus simple» qui soit égal à une forme normale. Soit M un terme représentant un programme, l'exécution de ce programme, ou un calcul, consiste donc à prendre ce terme de départ M et à trouver la forme normale N égale à M. La forme normale N est alors considérée comme le résultat du programme.

Pour obtenir une règle de calcul opérationnelle permettant de passer d'un terme à sa forme normale, on utilise l'axiome β dans une forme contrainte, c'est-à-dire unidirectionnelle, que l'on appelle β-réduction immédiate. Vue comme une relation sur les termes, la β-réduction est définie par :

$$\beta = \{((\lambda x.M)N,\ M[x := N]) \mid M,\ N \in \Lambda\}$$

Plus généralement, si on s'intéresse à une notion de réduction quelconque R, on peut définir la R-réduction immédiate (en une étape), notée $\rightarrow_R$, par :

$$\frac{(M,\ N) \in R}{M \rightarrow_R N} \qquad \frac{M \rightarrow_R N}{(M\,Z) \rightarrow_R (N\,Z)}$$

$$\frac{M \rightarrow_R N}{(Z\,M) \rightarrow_R (Z\,N)} \qquad \frac{M \rightarrow_R N}{\lambda x.M \rightarrow_R \lambda x.N}$$

La R-réduction, notée $\twoheadrightarrow_R$, est une relation entre deux termes obtenus l'un de l'autre en appliquant 0, une ou plusieurs R-réductions immédiates :

$$\frac{M \rightarrow_R N}{M \twoheadrightarrow_R N}$$

$$M \twoheadrightarrow_R M$$

$$\frac{M \twoheadrightarrow_R N \qquad N \twoheadrightarrow_R L}{M \twoheadrightarrow_R L}$$

La R-égalité est alors définie comme la fermeture symétrique et transitive de la relation R-réduction qui produit des classes d'équivalences sur les λ-termes.

Définition 3.6 *(R-égalité)*

$$\frac{M \twoheadrightarrow_R N}{M =_R N}$$

$$\frac{M =_R N}{N =_R M}$$

$$\frac{M =_R N \qquad N =_R L}{M =_R L}$$

Ces trois notions de R-réduction immédiate, de R-réduction et de R-égalité peuvent être appliquées à la notion de réduction β pour nous donner la β-réduction immédiate ($\rightarrow_\beta$), la β-réduction ($\twoheadrightarrow_\beta$) et la β-égalité ($=_\beta$). Le problème qui se pose maintenant est d'en tirer un mécanisme de calcul sur les programmes représentés par des λ-termes.

Pour que la β-réduction puisse être appliquée comme mécanisme de calcul sur les programmes représentés comme des λ-termes, il est important qu'elle soit indépendante du contexte. En particulier, si un terme M se réduit à un terme N, on désire que pour tout contexte $C[]$ à un trou, le terme $C[M]$ se réduise à $C[N]$. C'est ce que l'on appelle la notion de compatibilité de la relation :

Définition 3.7 ***(compatibilité)***

$R \subseteq \Lambda^2$ est compatible si $(M, M') \in R \Rightarrow (C[M], C[M']) \in R$ pour tout $M, M' \in \Lambda$ et tout contexte $C[]$ à un trou.

Proposition 3.1 *Les relations $\rightarrow_\beta$, $\twoheadrightarrow_\beta$ et $=_\beta$ sont toutes les trois compatibles.*

Structure générale de la preuve. *Pour $\rightarrow_\beta$, la compatibilité est une conséquence immédiate de la définition puisque les trois dernières clauses de la définition permettent de «pousser» le sous-terme réduit à l'endroit où l'on veut dans un terme.*

Pour $\twoheadrightarrow_\beta$, la preuve se fait par induction sur la définition, en examinant chacune des clauses. Les deux premières clauses servent de base d'induction : pour la première la compatibilité de $\twoheadrightarrow_\beta$ découle directement de la compatibilité de $\rightarrow_\beta$, alors que la seconde est triviale. Pour la troisième clause, $M \twoheadrightarrow_\beta L$ est la conséquence de $M \twoheadrightarrow_\beta N$ et $N \twoheadrightarrow_\beta L$. Or par hypothèse d'induction $C[M] \twoheadrightarrow_\beta C[N]$ et $C[N] \twoheadrightarrow_\beta C[L]$, donc $C[M] \twoheadrightarrow_\beta C[L]$.

La preuve pour $=_\beta$ se fait aussi par induction sur la définition.

Rassuré sur l'indépendance de la β-reduction par rapport au contexte, on peut maintenant définir un calcul d'un terme M comme la recherche d'une forme normale N telle que $M =_\beta N$. Pour expliciter davantage le mécanisme de calcul proposé pour normaliser les termes du λ-calcul, introduisons la notion d'expression réductible :

Définition 3.8 ***(redex ou expression réductible)***

Un β-redex est un terme M tel que $M \rightarrow_\beta N$ pour un certain terme N.

Cette notion d'expression réductible permet de proposer une deuxième définition, opérationnelle celle-là car liée au processus de réduction, d'une forme normale :

Définition 3.9 ***(forme normale)***

Un terme M est appelée une β-forme normale s'il ne contient aucun β-redex. Un terme N est une β-forme normale pour M si N est une β-forme normale et si $M =_\beta N$.

L'application de la β-réduction immédiate à un terme consiste donc à trouver un redex et à le transformer en sa forme normale. La proposition suivante établit qu'une étape de β-réduction immédiate est obtenue en trouvant un β-redex dans le terme courant et de le réduire pour obtenir un nouveau terme.

Proposition 3.2 *$M \to_\beta N \Leftrightarrow M \equiv C[P], N \equiv C[Q]$, et $(P,Q) \in \beta$ pour certains $P, Q \in \Lambda^2$.*

Preuve. *Partie $\Rightarrow$. La preuve est obtenue par induction sur la définition de $\to_\beta$.*
- *Cas $M \to_\beta N$ parce que $(M,N) \in \beta$: trivial en prenant $C[] = []$.*
- *Cas $M \to_\beta N$ parce que $M \equiv ZS$ et $N \equiv ZT$ et $S \to_\beta T$: l'hypothèse d'induction s'applique à $S \to_\beta T$ et donc il existe un contexte $C[]$ tel que $S \equiv C[P]$ et $T \equiv C[Q]$ avec $P \to_\beta Q$. Il suffit donc de prendre le contexte $ZC[]$ pour compléter la preuve.*
- *Les autres cas sont similaires.*

Partie $\Leftarrow$. Découle directement de la compatibilité de $\to_\beta$.

Ceci complète la vision de la β-réduction comme un calcul sur un programme. Le programme est représenté par un terme initial M, et son résultat N est une forme normale telle que $M =_\beta N$. Cette forme normale est obtenue en appliquant un série de pas de calcul qui consiste à trouver un redex dans le terme courant et à le réduire. Deux question fondamentales se posent alors :

1. Existe-t-il une forme normale pour tout terme M ?
2. Si cette forme normale existe, est-elle unique ?

Propriétés de la β-réduction

Avant de traiter ces deux questions un peu formellement, il est éclairant de comprendre informellement le problème qui est soulevé. À la question de savoir si tout terme a une forme normale, la réponse est non, car certains termes peuvent produire un phénomène de bouclage, c'est-à-dire que s'établit un cycle $M \twoheadrightarrow_\beta M$ dans le processus de réduction. Par exemple, le terme : $((\lambda x.xx)(\lambda x.xx))$

produit une boucle car l'application de la β-réduction immédiate redonne exactement le même terme (vérifiez !). Un tel terme n'a pas de forme normale. Conceptuellement, il n'est pas complètement surprenant que certains termes n'aient pas de forme normale dans le mesure où on s'attend intuitivement à ce qu'un formalisme assez puissant pour écrire toute fonction récursive, dont les fonctions avec une récursivité infinie, permette d'écrire un terme qui «boucle» sous la réduction.

La réponse à la seconde question est étroitement liée au fait que dans un terme M, il peut y avoir plusieurs redex susceptibles de faire l'objet d'une β-réduction. L'unicité de la forme normale d'un terme revient à se demander si le choix de la réduction à effectuer peut mener à plusieurs formes normales différentes. Le théorème fondamental ici est dû à Church et Rosser qui établit qu'un terme a au plus une forme normale (mais bien sûr il peut ne pas en avoir, comme on l'a vu pour les termes qui «bouclent»).

Les deux questions ne sont d'ailleurs pas totalement indépendantes puisque la multiplicité des redex dans un terme induit un choix de *stratégie de réduction* qui se définit par une règle déterminant pour tout terme quel redex doit être réduit. La stratégie de réduction peut également induire un comportement différent du calcul (du bouclage, par exemple, là où une autre stratégie conduit à une forme normale). Il existe deux grandes stratégies de choix

[2] où β est la β-réduction vue comme une relation définie précédemment

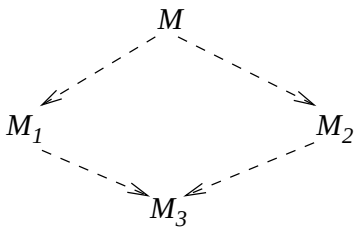
du sous-terme à réduire : l'appel par valeur consiste à toujours réduire les arguments d'une fonction avant l'application (les redex intérieurs avant les redex extérieurs), alors que l'appel par nom consiste à toujours réduire l'application de fonction avant de réduire les arguments (les redex extérieurs avant les redex intérieurs). On sait que certains termes bouclent sous la stratégie d'appel par valeur alors qu'on arrive à leur trouver une forme normale en appel par nom ! Informellement, la raison de ce comportement vient de la présence de termes qui bouclent dans les arguments de fonctions qui ne seront jamais évalués en appel par nom, alors qu'ils seraient toujours évalués en appel par valeur.

Essayons maintenant de formaliser un peu ces propriétés en débutant par une notion de convergence de la réduction des termes malgré une multiplicité de chemins possibles :

Définition 3.10 ***(Church–Rosser)***

Une notion de réduction R est dite Church–Rosser si $(\forall M, M_1, M_2)[M \twoheadrightarrow_R M_1 \wedge M \twoheadrightarrow_R M_2 \Rightarrow (\exists M_3)[M_1 \twoheadrightarrow_R M_3 \wedge M_2 \twoheadrightarrow_R M_3]]$.

La propriété Church–Rosser est aussi appelée la propriété du losange ou encore du parallèlogramme, en référence à la vision graphique des réductions où toute divergence dans la réduction d'un terme M en deux termes M_1 et M_2 sera ramenée vers un terme M_3 unique :



Théorème 3.2 ***(Théorème de Church–Rosser)*** *Soit R une notion de réduction Church–Rosser, alors* $M =_R N \Rightarrow (\exists Z)[M \twoheadrightarrow_R Z \wedge N \twoheadrightarrow_R Z]$.

Preuve. *Par induction sur la définition de $=_R$.*

- *Cas $M =_R N$ parce que $M \twoheadrightarrow_R N$: trivial, choisir $Z \equiv N$.*
- *Cas $M =_R N$ parce que $N =_R M$: trivial.*
- *Cas $M =_R N$ parce que $M =_R L$ et $L =_R N$: en appliquant au deux cas l'hypothèse d'induction, on trouve*

$$(\exists Z_1)[M \twoheadrightarrow_R Z_1 \wedge L \twoheadrightarrow_R Z_1]$$

et

$$(\exists Z_2)[L \twoheadrightarrow_R Z_2 \wedge N \twoheadrightarrow_R Z_2]$$

et par conséquence puisque $L \twoheadrightarrow_R Z_1$ et $L \twoheadrightarrow_R Z_2$, par l'hypothèse d'induction il existe un Z tel que $Z_1 \twoheadrightarrow_R Z$ et $Z_2 \twoheadrightarrow_R Z$. Par transitivité de $\twoheadrightarrow_R$, on a $M \twoheadrightarrow_R Z$ et $N \twoheadrightarrow_R Z$ (voir la figure 3.2).

Corollaire 3.1 *Soit R une relation Church–Rosser, alors :*

1. *si N est une R-forme normale de M alors $M \twoheadrightarrow_R N$*
2. *un terme peut avoir au plus une forme normale*

Preuve.

1. *Soit $M =_R N$, où N est une R-forme normale. Alors par le théorème $(\exists Z)[M \twoheadrightarrow_R Z \wedge N \twoheadrightarrow_R Z]$. Mais puisque N est une R-forme normale, par définition de $\twoheadrightarrow_R$, on doit avoir $N \equiv Z$.*

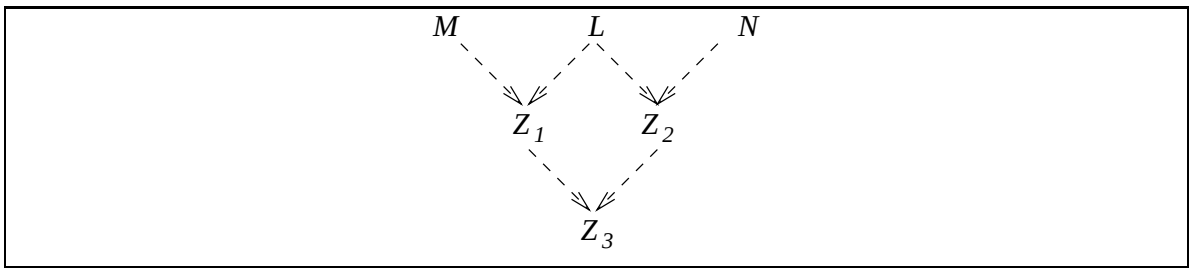


FIG. 3.2 – Convergence de la réduction dans le troisème cas du théorème de Church–Rosser.

2. *Soit N_1 et N_2 deux R-formes normales pour M. Alors $N_1 =_R M =_R N_2$ et donc $N_1 =_R N_2$. Par le théorème, on a $(\exists Z)[N_1 \twoheadrightarrow_R Z \wedge N_2 \twoheadrightarrow_R Z]$, et par définition de $\twoheadrightarrow_R$, on doit avoir $N_1 \equiv Z \equiv N_2$.*

Une approche qui saute maintenant aux yeux pour obtenir le résultat que l'on cherche, c'est-à-dire l'unicité de la forme normale, consiste tout simplement à montrer que la relation β définie plus haut est Church–Rosser. Malheureusement, ce n'est pas le cas. Un contre-exemple nous en est fourni par le terme : $(\lambda x.xx)((\lambda x.x)(\lambda x.x))$
qui par β-réduction immédiate nous donne deux termes divergents :
$((\lambda x.x)(\lambda x.x))((\lambda x.x)(\lambda x.x))$
$(\lambda x.xx)(\lambda x.x)$
qu'il est impossible de ramener à un unique terme par une étape de β-réduction immédiate.

Il est encore possible d'utiliser cette approche mais de façon un peu plus astucieuse. Certes, la relation β n'est pas Church–Rosser, mais pourrions-nous définir une nouvelle relation qui serait Church–Rosser et engendrant la même relation d'équivalence sur les termes que β, c'est-à-dire dans laquelle β serait incluse et dont la fermeture transitive serait la relation $\twoheadrightarrow_\beta$? Si c'est le cas, le résultat voulu découlerait du théorème de Church–Rosser et du corollaire 3.1.

Une telle relation existe bel et bien et est appelée «grande réduction». En gros, cette relation notée $\twoheadrightarrow_1$ a le pouvoir d'exécuter plusieurs β-reduction immédiate parallèle en une seule étape de calcul.

Définition 3.11 *(grande réduction)*

$$M \twoheadrightarrow_1 M$$

$$\frac{M \twoheadrightarrow_1 M'}{\lambda x.M \twoheadrightarrow_1 \lambda x.M'}$$

$$\frac{M \twoheadrightarrow_1 M' \qquad N \twoheadrightarrow_1 N'}{MN \twoheadrightarrow_1 M'N'}$$

$$\frac{M \twoheadrightarrow_1 M' \qquad N \twoheadrightarrow_1 N'}{(\lambda x.M)N \twoheadrightarrow_1 M'[x := N']}$$

Théorème 3.3 $(\forall M, N \in \Lambda)[M \to_\beta N \Rightarrow M \twoheadrightarrow_1 N]$.

Preuve. *Par induction sur la définition de $\to_\beta$.*
- *Cas $M \to_\beta N$ parce que $(M, N) \in \beta$: dans ce cas $M \equiv (\lambda x.P)Q$ et $N \equiv P[x := Q]$. Il suffit d'appliquer la quatrième clause de $\twoheadrightarrow_1$ avec $M \equiv M' \equiv P$) et $N \equiv N' \equiv Q$.*
- *Cas $SZ \to_\beta TZ$ parce que $S \to_\beta T$: appliquer la troisième clause de $\twoheadrightarrow_1$ avec $M \equiv S$ et $M' \equiv T$ et $N \equiv N' \equiv Z$. Par hypothèse d'induction, on aura $S \twoheadrightarrow_1 T$ et par définition on a $Z \twoheadrightarrow_1 Z$, donc $SZ \twoheadrightarrow_1 TZ$.*

– *Les deux autres cas sont similaires.*

Les quatre propriétes suivantes de la relation $\twoheadrightarrow_1$ peuvent être démontrées :

1. $M \twoheadrightarrow_1 M', N \twoheadrightarrow_1 N' \Rightarrow M[x := N] \twoheadrightarrow_1 M'[x := N']$.
2. $\lambda x.M \twoheadrightarrow_1 N \Rightarrow N \equiv \lambda x.M'$ avec $M \twoheadrightarrow_1 M'$.
3. $MN \twoheadrightarrow_1 L$ implique l'un ou l'autre de :
   – $L \equiv M'N'$ avec $M \twoheadrightarrow_1 M'$ et $N \twoheadrightarrow_1 N'$, ou
   – $M \equiv \lambda x.P, L \equiv P'[x := N']$ avec $P \twoheadrightarrow_1 P'$ et $N \twoheadrightarrow_1 N'$.
4. La relation $\twoheadrightarrow_1$ est Church–Rosser.

Nous allons maintenant démontrer la deuxième et la quatrième.

Lemme 3.1 *$\lambda x.M \twoheadrightarrow_1 N \Rightarrow N \equiv \lambda x.M'$ avec $M \twoheadrightarrow_1 M'$.*

Preuve. *Par induction sur la définition de $\twoheadrightarrow_1$.*
– *$N \equiv \lambda x.M$: trivial.*
– *$N \equiv \lambda x.M'$ avec $M \twoheadrightarrow_1 M'$: trivial.*
– *Les autres cas ne s'appliquent pas.*

Lemme 3.2 *La relation $\twoheadrightarrow_1$ est Church–Rosser.*

Preuve. *Par induction sur la définition de $M \twoheadrightarrow_1 M_1$ et en montrant que pour tout M_2 tel que $M \twoheadrightarrow_1 M_2$, il existe un M_3 tel que $M_1 \twoheadrightarrow_1 M_3$ et $M_2 \twoheadrightarrow_1 M_3$.*
– *Cas $M \equiv M_1$: choisir $M_3 \equiv M_2$.*
– *Cas $M \equiv \lambda x.P$ et $M_1 \equiv \lambda x.P'$ avec $P \twoheadrightarrow_1 P'$: Par le lemme précédent, M_2 doit être de la forme $\lambda x.P''$ avec $P \twoheadrightarrow_1 P''$. Par l'hypothèse d'induction, P' et P'' doivent avoir un contracté commun P''' et on choisit alors $M_3 \equiv \lambda x.P'''$.*
– *Cas $M \equiv PQ$ et $M_1 \equiv P'Q'$ avec $P \twoheadrightarrow_1 P'$ et $Q \twoheadrightarrow_1 Q'$: il y a deux cas à considérer :*
  – *$M_2 \equiv P''Q''$ avec $P \twoheadrightarrow_1 P''$ et $Q \twoheadrightarrow_1 Q''$ et alors par l'hypothèse d'induction P' et P'' (respectivement Q' et Q'') ont un contracté commun P''' (resp. Q''') et on choisit alors $M_3 \equiv P'''Q'''$.*
  – *$M_2 \equiv P_1''[x := Q'']$ avec $P \equiv \lambda x.P_1, P_1 \twoheadrightarrow_1 P_1''$ et $Q \twoheadrightarrow_1 Q''$: alors par le lemme précédent, $P' \equiv \lambda x.P_1'$ avec $P_1 \twoheadrightarrow_1 P_1'$. Par l'hypothèse d'induction appliquée à Q', Q'' et P_1', P_1'' il existe un contracté commun de M_1 et M_2 qui est $P_1'''[x := Q''']$.*
– *Cas $M \equiv (\lambda x.P)Q$, $M_1 \equiv P'[x := Q']$ et $P \twoheadrightarrow_1 P', Q \twoheadrightarrow_1 Q'$: encore une fois il y a deux cas à considérer selon que $M_2 \equiv P''[x := Q'']$ ou $M_2 \equiv (\lambda x.P'')Q''$. Dans les deux cas, l'argumentation est similaire au cas précédent.*

Cette première moitié du travail fait, il reste à établir que la réduction induite par $\twoheadrightarrow_1$ est bien «équivalente» à celle engendrée par la relation β. C'est l'objectif du théorème suivant :

Théorème 3.4 *$\twoheadrightarrow_\beta$ est la fermeture transitive de $\twoheadrightarrow_1$.*

Structure générale de la preuve. *Il s'agit de montrer que la fermeture symétrique de $\rightarrow_\beta$ est incluse dans $\twoheadrightarrow_1$, mais aussi que $\twoheadrightarrow_1$ est incluse dans $\twoheadrightarrow_\beta$ pour en conclure le résultat désiré.*

Retour sur le calcul par réduction

La grande réduction introduite précédemment est venue aidée la démonstration de l'unicité de la forme normale d'un terme de deux façons. D'une part, elle a donné du «parallélisme» à la β-réduction immédiate pour permettre de retrouver le contracté commun à deux termes

divergents par une application de la relation de grande réduction. De plus, et c'est probablement moins apparent, elle a permis de «récupérer» certains termes qui bouclent en ajoutant dans l'image de la relation des termes obtenus en passant par divers chemins nécessitant éventuellement une ou plusieurs applications de la β-réduction immédiate dans différents ordres. Par exemple, la grande réduction met en relation le terme $(\lambda x.y)((\lambda x.xx)(\lambda x.xx))$ avec le terme y en réduisant la fonction externe plutôt que d'essayer de réduire son argument.

Opérationnellement, la β-réduction immédiate apparaît comme un bon outil pour donner une étape élémentaire de calcul. Un calcul sera donc obtenu par application répétée de la β-réduction jusqu'à obtention d'une β-forme normale. L'exercice permis par la grande réduciton nous indique que l'ordre de réduction des redex a donc une grande importance. Ce qu'il ne nous dit pas, c'est le bon ordre de réduction à choisir pour réduire un terme à sa forme normale. Pour compléter une implantation, il faut définir une *stratégie de réduction*, c'est-à-dire une règle de choix du redex à réduire à chaque étape comme nous l'avons dit précédemment.

Certes, tout terme n'a pas nécessairement une forme normale. Un terme M est dit *normalisable* s'il possède une forme normale. Nous savons par le théorème de Church–Rosser et le fait que la grande réduction est Church–Rosser qu'alors la forme normale est unique. Un terme M est dit *fortement normalisable* s'il n'admet pas de réduction infinie. Bien sûr, il existe des termes qui sont normalisables mais pas fortement normalisables. Le terme $(\lambda x.y)((\lambda x.xx)(\lambda x.xx))$ en fournit un bon exemple.

Une stratégie de réduction sera dite *fortement normalisante* si elle permet toujours de trouver la forme normale d'un terme si celle-ci existe. Il existe deux grandes stratégies de réduction en λ-calcul : l'appel par valeur et l'appel par nom. En appel par valeur, les arguments des fonctions sont toujours réduits avant de réduire l'application qui les contiens. En appel par nom, la réduction de l'application se fait avant la réduction de ses arguments. La stratégie d'appel par nom est fortement normalisante, mais pas l'appel par valeur. En fait certains termes bouclent en appel par valeur alors qu'une forme normale est obtenue en appel par nom. Mieux, il a été démontré que si la réduction d'un terme en appel par nom est infinie, alors ce terme n'est pas normalisable.

Cela dit, le problème de la normalisation, c'est-à-dire de savoir si un terme possède une forme normale, est semi-décidable. Il existe un algorithme qui saura dire oui si une telle forme normale existe (la réduction en appel par nom si elle se termine), mais il n'est pas possible d'écrire un algorithme qui répondra non à coup sûr s'il n'y en a pas (la réduction en appel par nom sera infinie, la réponse risque donc de se faire attendre...). C'est un résultat en λ-calcul qui est analogue au fameux théorème d'arrêt des machines de Turing (ce théorême établit qu'on ne peut écrire un programme d'une machine de Turing qui prenne en entrée un autre programme P ainsi que son entrée E et qui dit si oui ou non si P boucle sur E).

δ-règles

Le λ-calcul fournit un outil complet pour aborder les fonctions calculables. Cependant, certains concepts ne peuvent être représentés que par un attirail très complexe. Par exemple, il existe un encodage des entiers sous forme de λ-termes et une réalisation des opérations sur les entiers sous forme de fonctions. Cependant cet encodage est très lourd à manipuler. De même, des structures de contrôle simples, comme l'alternative, peuvent être définies en termes de fonctions, mais cela rend les fonctions plutôt opaques.

Le moyen utilisé pour étendre le λ-calcul est de définir un certain nombre de *constantes*, représentant soit des valeurs primitives (exemple : 1, 2, 3, ...), soit des fonctions primitives

(exemple : +, -, ...). Ainsi, nous pouvons écrire des termes comme $((+\,x)\,y)$.

En utilisant simplement la β-réduction comme mécanisme de normalisation des termes, ces constantes vont rester dans le programme jusqu'à la fin et apparaître dans la forme normale obtenue. Pour traiter ces constantes, on ajoute de nouvelles règles de réduction spécifique appelées δ-règles. Par exemple, pour traiter les entiers introduits en λ-calcul, on introduit des règles de réduction permettant de dériver la constante 6 du terme $4 + 2$.

Cette pratique courante, que nous utiliserons dans les prochains chapitres sans l'expliciter mais simplement en construisant des λ-termes étendus, appelle une certaine prudence. En tant que règles de réduction, les δ-règles peuvent parfaitement briser la propriété Church–Rosser de la relation sur les λ-termes étendus et ainsi remettre en cause tout l'échafaudage théorique proposé ici. De façon générale, les δ-règles qui ne font intervenir que des termes qui contiennent des constantes ne posent pas trop de problèmes. Cependant, lorsque des sous-termes apparaissent dans les termes étendus et que le résultat d'une δ-règle peut produire un terme qui sera applicable (redex), un attention particulière doit être apportée pour ne pas perturber la normalisation des termes.

Exercices

1. Écrire en Scheme un programme qui normalise les λ-termes par β-réduction. Faites varier la stratégie de réduction et essayer votre normalisateur sur différents λ-termes.

Chapitre 4

Introduction à la sémantique dénotationnelle

La sémantique dénotationnelle est une technique pour donner la signification d'un programme en lui faisant correspondre un objet mathématique (entier, réel, fonction, ...) bien défini. Les pionniers de cette approche, parfois appelée sémantique mathématique ou encore sémantique de point-fixe, sont Christopher Strachey et Dana Scott. Dans ce chapitre, nous donnons un aperçu général de l'approche en mettant de côté les notions mathématiques sous-jacentes sur lesquelles nous reviendrons au prochain chapitre.

4.1 Syntaxe abstraite versus syntaxe concrète

Comme nous l'avons vu au chapitre 2, un langage est d'abord défini par sa syntaxe, qui elle-même fait intervenir, un alphabet, des signes de ponctuations, un vocabulaire et une grammaire. Un programme se présente d'abord comme une suite de caractères, le plus souvent conservée dans un fichier. Mais toute séquence de caractères ne forme pas un programme valide. Elle doit observer certaines régles dans la formation des mots, de phrases et du programme en entier.

Une première étape de la reconnaissance d'un programme valide (c'est-à-dire appartenant au langage) consiste à repérer les mots (constantes, identificateurs, mots-clés, etc.) et les signes de ponctuation pour vérifier s'ils appartiennent bien au langage. Ces mots et autres signes extérieurs, aussi appelés lexèmes, sont reconnus par l'analyseur lexicographique qui les classent en unités lexicales. Une fois identifiés, on doit vérifier que ces lexèmes, qui forment la séquence de mots et de signes extérieurs apparaissant dans le programme, respectent les régles de construction des phrases du langage (expressions artihmétiques et logiques, énoncés d'affectation et autre, fonctions et procédures, etc.).

La forme des phrases légales du langage est donnée par une grammaire. Une grammaire est définie par des règles de production qui servent à générer des phrases constituées de symboles terminaux en s'aidant de symboles auxiliaires appelés non-terminaux. Dans la grammaire d'un langage de programmation, les terminaux ne sont autres que les lexèmes identifiés durant la phase d'analyse lexicographique. Comme nous l'avons également vu au chapitre 2, les règles de production sont souvent données selon la notation dite Backus-Naur[1] Ces régles de production prescrivent des dérivations depuis le symbole non-terminal de départ, aussi apelé axiome,

[1] Des noms de ces deux inventeurs.

$$
\begin{array}{rcl}
E & ::= & E\texttt{+}T \mid E\texttt{-}T \mid T \\
T & ::= & T\texttt{*}F \mid F \\
F & ::= & I \mid \texttt{(}\ E\ \texttt{)} \\
I & ::= & n\text{on-spécifié}
\end{array}
$$

FIG. 4.1 – Grammaire des expressions arithmétiques

jusqu'à une séquence de terminaux formant une phrase légale. Chaque étape de dérivation comporte le remplacement d'un symbole non-terminal A de la proto-phrase courante[2] par la séquence de terminaux et non-terminaux apparaissant en partie droite d'une des règles de production définissant A.

Par exemple, la grammaire de la figure 4.1 engendre des expressions arithmétiques à partir de l'axiome E. Une dérivation de la phrase x+y*z est :

$$
\begin{array}{rcl}
E & \Rightarrow & E\texttt{+}T \Rightarrow T\texttt{+}T \Rightarrow F\texttt{+}T \Rightarrow I\texttt{+}T \Rightarrow \texttt{x+}T \Rightarrow \texttt{x+}T\texttt{*}F \Rightarrow \texttt{x+}F\texttt{*}F \Rightarrow \texttt{x+}I\texttt{*}F \\
 & \Rightarrow & \texttt{x+y*}F \Rightarrow \texttt{x+y*}I \Rightarrow \texttt{x+y*z}
\end{array}
$$

Pour décider si une phrase appartient au langage défini par une grammaire, il suffit de trouver une dérivation pour cette phrase depuis l'axiome. Ce problème de reconnaissance est celui de l'analyse syntaxique. Il est principalement concerné par l'identification des frontières entre phrases et sous-phrases en cours de balayage de la séquence linéaire des terminaux. À chaque étape de dérivation, deux choix doivent être faits : celui du non-terminal à remplacer et celui de la règle de production à utiliser pour faire le remplacement. Une grammaire pouvant produire plusieurs dérivations différentes pour une même phrase est dite ambigue.

Parce qu'elles sont liées à la reconnaissance des programmes et qu'elles travaillent sur leur aspect extérieur, les grammaires utilisées en analyse syntaxique sont dites grammaires concrètes. Le principal problème d'une grammaire concrète est l'élimination de toutes les ambiguités qui pourraient empêcher la reconnaissance des programmes. À ce titre, les grammaires concrètes forcent souvent l'introduction de terminaux et de non-terminaux dont le seul but est d'éviter les ambiguités. C'est le cas des non-terminaux T et F de la grammaire de la figure 4.1 dont le rôle consiste simplement à forcer une précédence précise entre les opérateurs.

Une fois ces difficultés surmontées et la dérivation obtenue, la structure hiérarchique du programme est obtenue. Une représentation classique des dérivations est l'arbre de dérivation. Un arbre de dérivation est un arbre étiqueté obtenu en créant le nœud racine étiqueté de l'axiome, puis en ajoutant comme fils de chaque nœud étiqueté d'un non-terminal A les nœuds correspondants aux terminaux et non-terminaux par lesquels A est remplacé dans la dérivation qui forment donc la partie droite d'une règle de production).

La notion de structure arborescente d'un programme est d'une importance cruciale. Passée l'étape de l'analyse syntaxique, on se rend compte qu'une telle représentation arborescente est de façon inhérente non-ambigüe ; elle est donc plus propice au raisonnement sur les programmes que la représentation sous forme de séquences de terminaux. Puisqu'un programme a

[2]On appelle proto-phrase une séquence de symbole terminaux et non-terminaux obtenue par un certain nombre d'étapes de dérivation depuis l'axiome ; une phrase ne contient que des terminaux.

$$e \quad ::= \quad e + e \mid e - e \mid e * e \mid id$$

FIG. 4.2 – Grammaire abstraite pour les expression arithmétiques

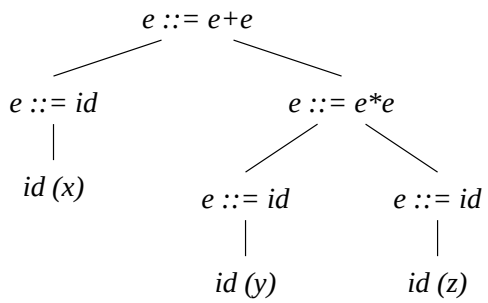


FIG. 4.3 – Arbre abstrait pour la phrase $id + id * id$

finalement une structure arborescente, pourquoi passer par une représentation linéaire (texte de programme) ?

Le principe de base de la syntaxe abstraite consiste précisément à voir un langage comme un ensemble d'arbres représentant les programmes admissibles. Puisque le syntaxe abstraite ne s'intéresse qu'à la structure compositionnelle d'un programme, les terminaux précis utilisées pour les représenter extérieurement perdent de leur pertinence. Par exemple, la structure d'un énoncé d'affectation consiste en un identificateur (la variable cible) et une expression (donnant la valeur à affecter) ; la forme extérieure de l'opérateur d'affectation nous est indifférent pour raisonner sur l'affectation.

Comment peut-on alors définir un langage comme l'ensemble des arbres formant des programmes admissibles ? En fait, de la même façon qu'un arbre peut être obtenu à partir d'une dérivation d'une phrase par une grammaire, une grammaire peut servir à définir la structure d'un ensemble d'arbres. À chaque type de nœud est associé une règle de production qui définit pour un certain type de nœud racine (partie gauche) et les types de nœuds fils (partie droite). À titre d'exemple, la figure 4.2 propose une grammaire abstraite pour nos expressions arithmétiques. Cette grammaire ne fait plus référence à des terminaux, mais plutôt à des types de nœuds (addition, soustraction et multiplication à deux nœuds fils, et identificateur sans fils). La figure 4.3 présente l'arbre abstrait correspondant à la phrase `x`+y+z (on suppose que chaque nœud de type *id* portera une indentification précise de la variable en jeu).

La sémantique dénotationnelle définit la sémantique d'un langage sur la base de sa syntaxe abstraite ; elle est donc dirigée par la syntaxe. Elle montre alors comment construire la dénotation associée à chaque phrase comme une fonction de la dénotation obtenue pour les sous-phrases. Ces deux caractéristiques de construction des sémantiques dénotationnelles sont à la base de la propriété de compositionnalité. Cette propriété est elle-même permet le raisonnement par induction structurelle sur les programmes, qui est la technique classique pour obtenir des preuves de propriétés. Nous y reviendrons. Par ailleurs, il est du ressort du sémanticien de choisir une syntaxe abstraite appropriée pour la définition de la sémantique d'un langage, sachant que certains choix peuvent rendre plus ou moins difficile l'obtention d'une sémantique.

4.2 Motivation

La sémantique opérationnelle d'un langage, par exemple impératif, est obtenue est définissant pour chaque énoncé c du langage comment il transforme un certain état initial de la mémoire σ en un état final σ', ce que l'on peut noter $\langle c, \sigma\rangle \rightarrow \sigma'$. En sémantique opérationnelle, la transformation de σ en σ' par c est définie en terme d'une séquence d'opérations de bas niveau qui rend difficile toute comparaison entre énoncés d'un même langage ou d'un langage à l'autre. Plutôt que de comparer les étapes de traitement associées à deux énoncés c_1 et c_2, il est plus judicieux de s'interroger sur leur résultat, c'est-à-dire que deux énoncés sont équivalents, noté $\sim$, si peu importe l'état initial, ils produisent le même état final :

$$c_1 \sim c_2 \; ssi \; (\forall \sigma, \sigma'. \langle c_1, \sigma\rangle \rightarrow \sigma' \Longleftrightarrow \langle c_2, \sigma\rangle \rightarrow \sigma')$$

Notons que $c_1 \sim c_2$ *ssi* :

$$\{(\sigma, \sigma') | \langle c_1, \sigma\rangle \rightarrow \sigma'\} = \{(\sigma, \sigma') | \langle c_2, \sigma\rangle \rightarrow \sigma'\}$$

c'est-à-dire que les deux énoncés définissent la même fonction partielle sur les états.

Une approche plus directe et plus abstraite pour capturer la sémantique des énoncés à leur équivalence près consiste donc à ne s'intéresser qu'à leur comportement en tant que fonctions partielles sur les états. Cette approche est celle de la sémantique dénotationnelle, où un énoncé c dénotera une fonction partielle $\mathcal{C}[\![c]\!] : \Sigma \rightarrow \Sigma$ où Σ représente l'ensemble des états possibles. $\mathcal{C}$ est un exemple de fonction sémantique. Les doubles crochets sont utilisés en sémantique dénotationnelle pour séparer ce qui est de l'ordre de la syntaxe de ce qui est de l'ordre de la sémantique. Soit **Enonces** le type des arbres syntaxiques traités, le type de la fonction $\mathcal{C}$ est donc plus exactement $\mathcal{C}$: **Enonces** $\rightarrow \Sigma \rightarrow \Sigma$. Outre qu'ils permettent de repérer facilement l'argument syntaxique (un arbre abstrait), ils soulignent un traitement légèrement différent de cet argument syntaxique par la fonction $\mathcal{C}$.

Cet argument syntaxique n'est pas évalué au sens traditionnel ; il peut contenir des méta-variables syntaxiques permettant de récupérer par filtrage des parties de la phrase syntaxique reçue en argument pour les utiliser ensuite dans la définition de la fonction $\mathcal{C}$. Par exemple, soit la fonction sémantique $\mathcal{A}$: **ExpA** $\rightarrow \Sigma \rightarrow I\!N$, nous écrirons $\mathcal{A}[\![a_0{+}a_1]\!]$ pour filtrer lors de l'appel de $\mathcal{A}$ un arbre de syntaxe abstraite représentant l'expression *5+3*4* pour récupérer le sous-arbre représentant l'expression *5* dans la méta-variable syntaxique a_0 et le sous-arbre représentant l'expression *3*4* dans la méta-variable syntaxique a_1.

4.3 Sémantique dénotationnelle d'un langage impératif simple

La figure 4.4 donne la syntaxe abstraite d'un langage impératif simple. Cette syntaxe comprend cinq types de phrases : les expressions arithmétiques (**ExpA**), les expressions logiques (**ExpB**), les énoncés (**Enonces**), les variables (**Ident**), et les nombres (**N**). Seule la forme des trois premiers est réellement intéressante ; la forme des autres n'est donc pas plus définie. Il est important de bien distinguer ici l'ensemble des arbres de syntaxe abstraite (triviaux) représentant des entiers, ici appelé **N**, de l'ensemble mathématique des entiers naturels $I\!N$; le premier est un type d'arbre, tel que nous pourrions le définir dans un langage de programmation, alors que le second est un ensemble d'objets mathématiques que nous allons justement essayer d'utiliser comme dénotations pour les phrases de type **N**.

$$
\begin{array}{lcl}
a & \in & \mathbf{ExpA} \\
b & \in & \mathbf{ExpB} \\
c & \in & \mathbf{Enonces} \\
x & \in & \mathbf{Ident} \\
n & \in & \mathbf{N}
\end{array}
$$

$$
\begin{array}{lcl}
a & ::= & n \mid x \mid a_0 + a_1 \mid a_0 - a_1 \mid a_0 \times a_1 \\
b & ::= & \mathit{true} \mid \mathit{false} \mid a_0 = a_1 \mid a_0 \leq a_1 \mid \neg b \mid b_0 \vee b_1 \mid b_0 \wedge b_1 \\
c & ::= & \mathit{skip} \mid x := a \mid c_0 \,;\, c_1 \mid \mathit{if}\ b\ \mathit{then}\ c_0\ \mathit{else}\ c_1 \mid \mathit{while}\ b\ \mathit{do}\ c
\end{array}
$$

FIG. 4.4 – Grammaire abstraite pour un langage impératif simple

La sémantique dénotationnelle de notre langage impératif simple doit faire correspondre à chaque objet syntaxique une dénotation dans un ensemble d'objets mathématiques appropriés. Il faut donc définir un ensemble[3] de dénotations pour chacun des types de phrases du langage. Pour les nombre, l'ensemble évident est l'ensemble $\mathbb{N}$, et la correspondance entre les entiers syntaxiques et les entiers sémantiques est donnée par la fonction $\mathcal{N} : \mathbf{N} \rightarrow \mathbb{N}$ que nous n'explicitons pas plus avant.

Pour comprendre la dénotation les variables, il faut aussi parler de la notion d'état. La variable impérative n'est qu'une abstraction sur les adresses des emplacements mémoire choisis pour contenir leur valeur. La dénotation d'une phrase de type **Ident** est donc simplement un ensemble d'adresses mémoire que nous appellerons **Loc**. Pour obtenir la dénotation d'une variable, il suffit de disposer d'une simple correspondance 1-1 entre l'ensemble **Ident** et un sous-ensemble de **Loc** ; nous ne nous étendons pas plus sur cela, sauf pour dire qu'il arrive souvent, pour plus de simplicité dans les sémantiques dénotationnelles, que l'on assimile **Ident** et **Loc** à un seul et même ensemble.

La mémoire peut simplement être vue comme une fonction des adresses dans les valeurs associables aux variables. Dans le cadre de notre langage impératif simple, une variable ne peut contenir qu'un entier. L'ensemble des mémoires, noté Σ, sera donc constitué de fonctions $\sigma : \mathbf{Loc} \rightarrow \mathbb{N}$. Donc, pour $l = \mathcal{L}[\![x]\!]$, le contenu de la mémoire σ pour la variable x est donné par $\sigma(l)$.

Considérons maintenant les phrases plus complexes, et d'abord les expressions arithmétiques. Nous avons tous un modèle intuitif de ce que peut engendrer l'exécution d'une expression arithmétique. Outre le fait qu'elle doit retourner comme résultat un nombre, la nature impérative de notre langage laisse supposer que lorsque nous rencontrerons une variable, il faudra en obtenir la valeur et donc le contenu de l'emplacement mémoire correspondant. La sémantique d'une expression arithmétique doit donc tenir compte d'un élément de contexte représenté par la mémoire. La dénotation des expressions arithmétiques est donnée par la

[3]En fait, ce seront des structures mathématiques légèrement plus compliquées, comme nous le verrons au prochain chapitre.

fonction $\mathcal{A}$ suivante :

$$\begin{aligned}
\mathcal{A}[\![n]\!] &= \{(\sigma, \mathcal{N}[\![n]\!]) \mid \sigma \in \Sigma\} \\
\mathcal{A}[\![x]\!] &= \{(\sigma, \sigma(\mathcal{L}[\![x]\!])) \mid \sigma \in \Sigma\} \\
\mathcal{A}[\![a_0 + a_1]\!] &= \{(\sigma, n_0 + n_1) \mid (\sigma, n_0) \in \mathcal{A}[\![a_0]\!] \ \& \ (\sigma, n_1) \in \mathcal{A}[\![a_1]\!]\} \\
\mathcal{A}[\![a_0 - a_1]\!] &= \{(\sigma, n_0 - n_1) \mid (\sigma, n_0) \in \mathcal{A}[\![a_0]\!] \ \& \ (\sigma, n_1) \in \mathcal{A}[\![a_1]\!]\} \\
\mathcal{A}[\![a_0 \times a_1]\!] &= \{(\sigma, n_0 \times n_1) \mid (\sigma, n_0) \in \mathcal{A}[\![a_0]\!] \ \& \ (\sigma, n_1) \in \mathcal{A}[\![a_1]\!]\}
\end{aligned}$$

Il est important de remarquer que chacun des signes $+, -, \times$ à gauche des équations représente un type de noeuds de l'arbre de syntaxe abstraite alors que les mêmes signes en partie droite des mêmes équations désignent plutôt les fonctions d'addition, de soustraction et de multiplication sur les entiers. Ce genre de raccourcis de notation entre la sphère syntaxique et la sphère sémantique est monnaie courante pour simplifier les sémantiques et généralement ils n'entraînent pas de réelle confusion. On obtient donc des expressions du genre :

$$\mathcal{A}[\![3 + 5]\!]\sigma = \mathcal{A}[\![3]\!]\sigma + \mathcal{A}[\![5]\!]\sigma = 3 + 5 = 8$$

comme il se doit.

Chacun des ensembles définis par la fonction $\mathcal{A}$ représente en fait lui-même une fonction $\Sigma \to I\!N$. Pour représenter de telles fonctions, on utilise la plupart du temps la notation du λ-calcul. Ceci donne une définition équivalente de $\mathcal{A}$[4] :

$$\begin{aligned}
\mathcal{A}[\![n]\!] &= \lambda\sigma.\mathcal{N}[\![n]\!] \\
\mathcal{A}[\![x]\!] &= \lambda\sigma.\sigma(\mathcal{L}[\![x]\!]) \\
\mathcal{A}[\![a_0 + a_1]\!] &= \lambda\sigma.(\mathcal{A}[\![a_0]\!]\sigma + \mathcal{A}[\![a_1]\!]\sigma) \\
\mathcal{A}[\![a_0 - a_1]\!] &= \lambda\sigma.(\mathcal{A}[\![a_0]\!]\sigma - \mathcal{A}[\![a_1]\!]\sigma) \\
\mathcal{A}[\![a_0 \times a_1]\!] &= \lambda\sigma.(\mathcal{A}[\![a_0]\!]\sigma \times \mathcal{A}[\![a_1]\!]\sigma)
\end{aligned}$$

La sémantique des expressions logiques exhibe une structure très similaire à celle des expressions arithmétiques. Le résultat d'une expression logique doit être une valeur de vérité, vraie ou fausse. La dénotation fera donc intervenir l'ensemble $\mathbf{T} = \{true, false\}$ des valeurs de vérités, muni des opérations de négation ($\neg$), de disjonction ($\vee$), et de conjonction ($\wedge$) classiques de la logique mathématique.

À première vue, on serait tenté de dire qu'il est inutile de tenir compte de la mémoire du contexte d'évaluation des expressions logiques, car nous n'avons pas introduit de variables logiques. Pourtant, une expression logique peut être constituée d'expression arithmétiques, ce qui exige de tenir compte du contexte. En notation du λ-calcul, nous introduisons la fonction $\mathcal{B} : \Sigma \to \mathbf{T}$:

$$\begin{aligned}
\mathcal{B}[\![\mathbf{true}]\!] &= \lambda\sigma.true \\
\mathcal{B}[\![\mathbf{false}]\!] &= \lambda\sigma.false \\
\mathcal{B}[\![a_0 = a_1]\!] &= \lambda\sigma.(\mathcal{A}[\![a_0]\!]\sigma = \mathcal{A}[\![a_1]\!]\sigma) \\
\mathcal{B}[\![a_0 \leq a_1]\!] &= \lambda\sigma.(\mathcal{A}[\![a_0]\!]\sigma \leq \mathcal{A}[\![a_1]\!]\sigma) \\
\mathcal{B}[\![\neg b]\!] &= \lambda\sigma.\neg\mathcal{B}[\![b]\!]\sigma \\
\mathcal{B}[\![b_0 \ \vee \ b_1]\!] &= \lambda\sigma.\mathcal{B}[\![b_0]\!]\sigma \ \vee \ \mathcal{B}[\![b_1]\!]\sigma \\
\mathcal{B}[\![b_0 \ \wedge \ b_1]\!] &= \lambda\sigma.\mathcal{B}[\![b_0]\!]\sigma \ \wedge \ \mathcal{B}[\![b_1]\!]\sigma
\end{aligned}$$

[4]Certains auteurs ajoutent systématiquement les types des arguments à chaque définition de λ-expression, ce qui est extrêmement lourd à la longue. De notre point de vue, il suffit de connaître le type de la fonction définie, qui induit le type de la λ-expression définissante.

La définition de la fonction $\mathcal{C}[\![c]\!]$ pour les énoncés pose plus de difficultés. Comme nous l'avons vu précédemment, la dénotation des énoncés peut être capturée comme une fonction des états vers les états, c'est-à-dire qu'un énoncé n'a pour rôle que de changer l'état de la mémoire. La fonction $\mathcal{C}$ doit donc avoir pour type $\mathbf{Enonces} \to \Sigma \to \Sigma$. Sur cette lancée, il est relativement simple de définir $\mathcal{C}$ pour les énoncés nul, affectation, séquence et alternative :

$$\begin{aligned}
\mathcal{C}[\![skip]\!] &= \lambda\sigma.\sigma \\
\mathcal{C}[\![x := a]\!] &= \lambda\sigma.\sigma[(\mathcal{A}[\![a]\!]\sigma)/\mathcal{L}[\![x]\!]\,] \\
\mathcal{C}[\![c_0\ ;\ c_1]\!] &= \mathcal{C}[\![c_0]\!] \circ \mathcal{C}[\![c_1]\!] \\
\mathcal{C}[\![if\ b\ then\ c_0\ else\ c_1]\!] &= \lambda\sigma.\mathbf{if}\ \mathcal{B}[\![b]\!]\sigma\ \mathbf{then}\ \mathcal{C}[\![c_0]\!]\sigma\ \mathbf{else}\ \mathcal{C}[\![c_1]\!]\sigma
\end{aligned}$$

où $\sigma[K/x]$ est la fonction partout identique à σ sauf éventuellement au point x où elle vaut K.

Pour l'énoncé de répétition *while b do c*, comment trouver une dénotation qui ne dépende que des sous-phrases b et c ? Il est tentant d'utiliser l'équivalence suivante :

$$while\ b\ do\ c\ \sim\ if\ b\ then\ c\ ;while\ b\ do\ c\ else\ skip$$

En effet, si le test de la répétition est vrai, on exécute une fois le corps de la boucle puis on recommence l'exécution de la boucle. Si le test est faux, on passe à la suite. Selon cette idée, nous obtiendrions la définition suivante pour $\mathcal{C}$:

$$\mathcal{C}[\![while\ b\ do\ c]\!] = \lambda\sigma.\mathbf{if}\ \mathcal{B}[\![b]\!]\sigma\ \mathbf{then}\ \mathcal{C}[\![while\ b\ do\ c]\!](\mathcal{C}[\![c]\!]\sigma)\ \mathbf{else}\ \sigma$$

Le problème avec cette définition est que cette sémantique de la répétition n'est pas définie uniquement en fonction de ses sous-phrases mais aussi d'elle-même. Cette définition n'est donc pas compositionnelle, une condition essentielle pour toute sémantique dénotationnelle.

4.4 Points fixes

Revenons sur la notation ensembliste des fonctions. Soit $w = while\ b\ do\ c$, l'équivalence précédente se réécrit :

$$w\ \sim\ \mathbf{if}\ b\ \mathbf{then}\ c\ ;\ w\ \mathbf{else}\ skip$$

En sémantique dénotationnelle, cette équivalence, nous l'avons dit, s'interprète comme le fait que les deux phrases doivent dénoter la même fonction partielle sur les états, c'est-à-dire :

$$\mathcal{C}[\![w]\!] = \mathcal{C}[\![\mathbf{if}\ b\ \mathbf{then}\ c\ ;\ w\ \mathbf{else}\ skip]\!]$$

Nous devons donc avoir :

$$\begin{aligned}
\mathcal{C}[\![w]\!] &= \{(\sigma,\sigma') \mid \mathcal{B}[\![b]\!]\sigma = true\ \&\ (\sigma,\sigma') \in \mathcal{C}[\![c\ ;\ w]\!]\} \\
&\quad \cup \{(\sigma,\sigma) \mid \mathcal{B}[\![b]\!]\sigma = false\} \\
&= \{(\sigma,\sigma') \mid \mathcal{B}[\![b]\!]\sigma = true\ \&\ (\sigma,\sigma') \in \mathcal{C}[\![w]\!] \circ \mathcal{C}[\![c]\!]\} \\
&\quad \cup \{(\sigma,\sigma) \mid \mathcal{B}[\![b]\!]\sigma = false\}
\end{aligned}$$

Posons $\phi = \mathcal{C}[\![w]\!]$, $\beta = \mathcal{B}[\![b]\!]$ et $\gamma = \mathcal{C}[\![c]\!]$. Nous cherchons alors une fonction partielle telle que :

$$\begin{aligned}
\phi &= \{(\sigma,\sigma') \mid \beta(\sigma) = true\ \&\ (\sigma,\sigma') \in \phi \circ \gamma\} \\
&\quad \cup \{(\sigma,\sigma) \mid \beta(\sigma) = false\}
\end{aligned}$$

Puisque ϕ apparaît des deux côtés de l'équation, il faut trouver une façon de calculer une fonction ϕ qui vérifie l'égalité. L'approche classique pour résoudre de telles équations récursives est celle du point fixe. À partir d'une approximation ϕ_i de la fonction ϕ, il faut trouver un moyen de tourner l'équation en une règle permettant de calculer une «meilleure» approximation ϕ_{i+1}. Dans de bonnes conditions, l'application répétée de ce processus converge vers ϕ. La règle d'amélioration dans notre cas est obtenue en voyant l'équation précédente comme l'application d'un opérateur Γ :

$$\begin{aligned}\Gamma(\phi) &= \{(\sigma,\sigma') \mid \beta(\sigma) = true \;\&\; (\sigma,\sigma') \in \phi \circ \gamma\} \\ &\quad \cup \{(\sigma,\sigma) \mid \beta(\sigma) = false\} \\ &= \{(\sigma,\sigma') \mid \exists\, \sigma''.\beta(\sigma) = true \;\&\; (\sigma,\sigma'') \in \gamma \;\&\; (\sigma'',\sigma') \in \phi\} \\ &\quad \cup \{(\sigma,\sigma) \mid \beta(\sigma) = false\}\end{aligned}$$

et alors nous cherchons un point fixe de l'équation :

$$\phi = \Gamma(\phi)$$

L'approche du calcul du point fixe par opérateur s'applique à ce cas. Posons d'abord $\phi_0 = \{(\sigma,\sigma) | \beta(\sigma) = false\}$, c'est-à-dire toutes les paires d'états dans lequels le test est faux et qui ne nécessiteront donc pas de passer dans le corps de la boucle. L'application de l'opérateur Γ à ϕ_0 produit un nouvel ensemble ϕ_1 :

$$\begin{aligned}\phi_1 &= \{(\sigma,\sigma') \mid \exists\, \sigma''.\beta(\sigma) = true \;\&\; (\sigma,\sigma'') \in \gamma \;\&\; (\sigma'',\sigma') \in \phi_0\} \\ &\quad \cup \{(\sigma,\sigma) \mid \beta(\sigma) = false\} \\ &= \{(\sigma,\sigma') \mid \exists\, \sigma''.\beta(\sigma) = true \;\&\; (\sigma,\sigma'') \in \mathcal{C}[\![c]\!] \;\&\; (\sigma'',\sigma') \in \phi_0\} \\ &\quad \cup \{(\sigma,\sigma) \mid \beta(\sigma) = false\}\end{aligned}$$

Autrement dit, ϕ_1 contient ϕ_0 plus l'ensemble des paires (σ,σ') telles que le test est vrai dans σ mais faux dans σ', c'est-à-dire les états dans lesquels un seul tour de boucle sera nécessaire.

En poursuivant ce processus, on construit ϕ_2 qui va contenir toutes les paires d'états dans lesquels au plus deux tours de boucles seront nécessaires, et ainsi de suite. Ce processus converge, au sens de la théorie des points fixes, vers l'énsemble des paires (σ,σ') où au plus $n \to \infty$ tours de boucles seront nécessaires, pour tout n fini.

Ce genre de dénotations obtenues par calcul de points fixes apparaissent dans toutes sémantiques dénotationnelles comportant des phrases non-triviales (itération, récursivité), ou si vous préférez dans tout langage qui se veut Turing-équivalent. Cette proéminence explique que l'on appelle également la sémantique dénotationnelle la sémantique des points fixes (en anglais «*fixpoint semantics*»). Nous y reviendrons plusieurs fois dans les prochains chapitres.

4.5 Éléments constituant une sémantique dénotationnelle

L'objectif de toute sémantique dénotationnelle est de donner une signification aux programmes écrits dans un certain langage de programmation. À ce titre, une sémantique s'intéresse d'abord à des programmes, définis par leur syntaxe abstraite. Ainsi, toute sémantique s'attachera dans un premier temps à définir précisément et complètement ce que l'on appelle les domaines syntaxiques représentant les programmes du langage traité. Comme nous l'avons

vu à la section précédente, les domaines syntaxiques sont des ensembles d'arbres dont la forme est le plus souvent définie par une grammaire abstraite.

L'intuition essentielle de la sémantique dénotationnelle consiste à associer à toute phrase d'un programme une dénotation sous la forme d'un objet mathématique (entier, réel, fonction, etc.). Ainsi, après la définition de la syntaxe abstraite, la sémantique définit les dénotations utilisées, en précisant les ensembles[5] et les opérations dont ils sont munis. Il s'agit des domaines sémantiques.

La troisiéme partie de la sémantique sera formée des fonctions sémantiques, dont la principale caractéristique est de posséder un argument de type syntaxique. Pour chaque domaine syntaxique, il faudra définir une fonction sémantique. Chaque fonction sémantique est elle-même définie par un nombre fini d'équations sémantiques correspondant à chaque type de phase du domaine syntaxique traité. Par exemple, dans la sémantique de la section précédente, nous avions une fonction sémantique pour chacun des domaines syntaxiques : expressions artihmétiques, expressions logiques et énoncés. Dans le cas de la fonction $\mathcal{A}$, nous avons défini des équations sémantiques pour chaque type d'expressions arithmétiques : constante, variable, addition, soustraction, et multiplication.

Le rôle des équations sémantiques est d'indiquer comment associer une dénotation aux phrases dans un programme. Donner une sémantique dénotationnelle d'un langage consiste donc à définir pour chaque type de phrase comment calculer la dénotation associée à une phrase particuliére de ce type. En effet, le calcul d'une dénotation se réalise nécessairement sur une phrase particulière d'un programme. Par exemple, lorsque nous avons défini l'équation sémantique pour les phrases de type expressions arithmétiques d'addition, nou avons donné une règle permettant de prendre une expression d'addition et d'obtenir sa dénotation.

Enfin, une sémantique dénotationnelle peut comporter au besoin des définitions de fonctions auxiliaires permettant de mieux structurer l'ensemble de la définition. Ces fonctions ne comportent pas d'argument ni de résultats de types syntaxiques, mais uniquement de types sémantiques. Le langage du λ-calcul étant plutôt frustre, on recourt assez facilement à des telles fonctions auxiliaires.

[5] Nous nous en tenons au mot ensemble pour l'instant ; nous verrons au prochain chapitre qu'il faudra en réalité faire appel à des entités comportant un peu plus de structure, c'est-à-dire des domaines.

Chapitre 5

Éléments théoriques sous-jacents à la sémantique dénotationnelle

Au chapitre précédent, nous avons survolé les principaux concepts participants à une sémantique dénotationnelle. Ce faisant, nous avons touchés à certaines notions théoriques sur lesquelles est fondée la sémantique dénotationnelle : les points fixes et les domaines. Dans ce chapitre, nous revenons de manière plus formelle sur ces notions.

5.1 Points fixes et ordres partiels complets

Au chapitre précédent, nous avons vu comment certaines dénotations impliquent la résolution d'équations récursives faite par approximations successives selon une approche impliquant un calcul de point fixe. Nous nous sommes alors basés sur une théorie des opérateurs sur les ensembles et de leurs points fixes.

5.1.1 Ensembles définis par des règles, opérateurs et points fixes

Un ensemble peut être défini par des règles. Les instances de règles ont la forme $(\varnothing/x)$ ou la forme $(\{x_1, x_2, ..., x_n\}/x)$. Une instance de règle s'applique à la dérivation d'éléments nouveaux à partir d'éléments précédents. L'interprétation de la première forme d'instances de règles est au'on peut dériver x à partir de l'ensemble vide d'éléments précédents, alors que la seconde permet de dériver le nouvel élément x à partir des éléments x_1, x_2, ..., x_n.

Étant donné R un ensemble d'instances de règles, on note I_R l'ensemble défini par R contenant précisément les éléments dérivables à partir des intances de règles. Les ensembles définis par des règles sont très utiles car ils permettent d'appliquer une approche inductive pour les preuves de propriétés sur laquelle nous reviendrons plus loin. Le principe de base est le suivant. Soit I_R un ensemble engendré par un ensemble de règles R. Soit P une propriété. Alors $\forall x \in I_R.P(x)$ ssi pour toutes les instances de règles (X/y) dans R pour lesquelles $X \subseteq I_R$ on a $(\forall x \in X.P(x)) \Rightarrow P(y)$.

Un ensemble Q est dit fermé sous l'ensemble d'instances de règles R, ou simplement R-fermé, ssi pour toute instance de règle $(X/y) \in R$, $X \subseteq Q \Rightarrow y \in Q$. Un résultat important est que I_R est le plus petit ensemble R-fermé, ou plus précisément : étant donné un ensemble d'instances de règles R,

1. I_R est R-fermé, et

2. si Q est un ensemble R-fermé, alors $I_R \subseteq Q$.

Une autre façon de voir un ensemble défini par des règles est de voir R comme déterminant un opérateur $\hat{R}$ sur les ensembles qui, étant donné un ensemble B construit l'ensemble :

$$\hat{R}(B) = \{y \mid \exists X \subseteq B.(X/y) \in R\}$$

L'opérateur $\hat{R}$ fournit un moyen pour construire l'ensemble I_R. $\hat{R}$ est un opérateur monotone, au sens où $A \subseteq B \Rightarrow \hat{R}(A) \subseteq \hat{R}(B)$. En appliquant l'opérateur $\hat{R}$ répétitivement à partir de l'ensemble vide, on obtient une séquence d'ensembles :

$$\begin{aligned} A_0 &= \hat{R}^0(\varnothing) = \varnothing \\ A_1 &= \hat{R}^1(\varnothing) = \hat{R}(\varnothing) \\ A_2 &= \hat{R}^2(\varnothing) = \hat{R}(\hat{R}(\varnothing)) \\ &\vdots \\ A_n &= \hat{R}^n(\varnothing), \\ &\vdots \end{aligned}$$

Par la monotonicité de l'opérateur $\hat{R}$, cette séquence d'ensembles forme en fait une chaîne $A_0 \subseteq A_1 \subseteq A_2 \cdots \subset A_n \subseteq \ldots$. En effet, on a nécessairement que $\varnothing \subseteq \hat{R}(\varnothing)$, de par la nature de l'ensemble vide qui est inclus dans tout ensemble. La monotonicité de $\hat{R}$ assure qu'en l'appliquant aux deux côtés de cette inclusion, on obtient $\hat{R}(\varnothing) \subseteq \hat{R}(\hat{R}(\varnothing))$. On peut appliquer ce raisonnement entre chacun des groupes de deux étapes consécutives ci-haut pour obtenir la chaîne d'inclusion des A_i. Si on pose $A = \bigcup_{n\in\omega} A_n$, on a :

1. A est R-fermé.
2. $\hat{R}(A) = A$.
3. A est le plus petit ensemble R-fermé.

Ainsi, $A = I_R$, ou dit autrement I_R est le point fixe de l'opérateur $\hat{R}$, noté $fix(\hat{R})$ et obtenu par la construction :

$$fix(\hat{R}) \quad =_{def} \quad \bigcup_{n\in\omega} \hat{R}^n(\varnothing)$$

5.1.2 Ordres partiels complets

Bien que nous ne les ayions pas faites explicitement, les preuves des théorèmes précédents sur les points fixes d'opérateurs sur les ensembles sont basées sur une propriété essentielle de R, sa finitude. En pratique, peu de définitions récursives s'avèrent exprimables en termes de points fixes d'operateurs sur les ensembles. Pour plus de généralité de façon à servir de fondement à la sémantique dénotationnelle, il faut introduire la notion d'ordre partiel complet et de fonction continue. En fait, il s'agit de généraliser l'ordre d'inclusion entre les ensembles à des cas plus complexes.

Définition 5.1 ***(ordre partiel)***

Un ordre partiel, aussi o.p., est un ensemble P sur lequel est définie une relation binaire $\sqsubseteq$ qui est :

1. *réflexive : $\forall p \in P.p \sqsubseteq p$*
2. *transitive : $\forall p, q, r \in P.p \sqsubseteq q \wedge q \sqsubseteq r \Rightarrow p \sqsubseteq r$*

3. antisymétrique : $\forall p, q \in P.p \sqsubseteq q \wedge q \sqsubseteq p \Rightarrow p = q$

Pour obtenir des constructions similaires aux points fixes des opérateurs sur les ensembles, on doit aussi nous munir d'une notion de plus petite borne supérieure (PPBS) sur les ordres partiels.

Définition 5.2 ***(borne supérieure)***

Pour un ordre partiel $(P, \sqsubseteq)$ *et un sous-ensemble* $X \subseteq P$*, un élément* p *est une borne supérieure de* X *ssi* $\forall q \in X.q \sqsubseteq p$*.*

On dit que p *est une plus petite borne supérieure de* X *ssi*

1. p *est une borne supérieure de* X*, et*

2. pour toutes les bornes supérieures q *de* X*,* $p \sqsubseteq q$*.*

On note $\bigsqcup X$ *la PPBS de* X *lorsqu'elle existe, et on écrit* $\bigsqcup\{d_1, \ldots, d_m\}$ *comme* $d_1 \sqcup d_2 \sqcup \cdots \sqcup d_m$*.*

Définition 5.3 ***(ordre partiel complet)***

Soit $(D, \sqsubseteq_D)$ *un ordre partiel.*

- *Une* ω*-chaîne de l'ordre partiel est une chaîne croissante* $d_1 \sqsubseteq_D d_2 \sqsubseteq_D \cdots \sqsubseteq_D d_n \sqsubseteq_D \ldots$ *d'éléments de l'ordre partiel.*
- *L'ordre partiel* $(D, \sqsubseteq_D)$ *est un ordre partiel complet, aussi o.p.c., si toute* ω*-chaîne croissante* $\{d_n \mid n \in \omega\}$ *possède une PPBS* $\bigsqcup\{d_n \mid n \in \omega\}$*, aussi notée* $\bigsqcup_{n\in\omega} d_n$*.*
- $(D, \sqsubseteq_D)$ *est un o.p.c. «avec bottom» s'il est un o.p.c. et qu'il possède un plus petit élément* $\bot_D$ *appelé «bottom».*

Muni de ces notions d'ordres partiels complets, il est nécessaire de définir une notion correspondant aux opérateurs de notre approche précédente. Cette notion est celle de fonction continue.

Définition 5.4 ***(fonction continue)***

Une fonction $f : D \to E$ *entre deux o.p.c.* D *et* E *est monotone ssi* $\forall d, d' \in D.d \sqsubseteq d' \Rightarrow f(d) \sqsubseteq f(d')$*.*

Une telle fonction est continue ssi elle est monotone et que pour toute chaîne $d_1 \sqsubseteq_D d_2 \sqsubseteq_D \cdots \sqsubseteq_D d_n \sqsubseteq_D \ldots$ *dans* D *on a :*

$$\bigsqcup_{n\in\omega} f(d_n) = f(\bigsqcup_{n\in\omega} d_n)$$

Une conséquence importante de cette définition est que toute fonction continue d'un o.p.c. «avec bottom» vers lui-même a un plus petit point fixe.

Définition 5.5 ***(point fixe et pré-point fixe)***

Soit $f : D \to D$ *une fonction continue sur un o.p.c.* D*, un point fixe de* f *est un élément* d *de* D *tel que* $f(d) = d$*. Un pré-point fixe est un élément* d *de* D *tel que* $f(d) \sqsubseteq d$*.*

Théorème 5.1 ***(point fixe)***

Soit $f : D \to D$ *une fonction continue sur un o.p.c.* D *avec bottom. Définissons :*

$$fix(f) = \bigsqcup f^n(\bot)$$

Alors $fix(f)$ *est un point fixe de* f *et le plus petit pré-point fixe de* f*, c'est-à-dire :*

1. $f(fix(f)) = fix(f)$, *et*
2. *si* $f(d) \sqsubseteq d$ *alors* $fix(f) \sqsubseteq d$.

Par conséquence, $fix(f)$ *est le plus petit point fixe de* f.

5.1.3 Retour sur la sémantique dénotationnelle

En sémantique dénotationnelle, les «types de données» nécessaires à la définition d'une sémantique seront modélisés à l'aide d'ordres partiels complets et les calculs seront modélisés par des fonctions continues sur ceux-ci. L'intuition derrière cette utilisation de ces notions formelles sera de voir les éléments des o.p.c. comme une information sur la dénotation d'une phrase et l'ordre $x \sqsubseteq y$ comme une notion d'approximation de y par x. Dans cette intuition, l'élément $\perp$ dénotera l'absence totale d'information sur la dénotation. Il faut noter que tout ensemble ordonnée par la relation d'identité constitue un o.p.c. Si on lui adjoint un élément $\perp$ plus petit que tous les autres éléments de l'ensemble, on obtient un o.p.c. avec bottom appelé o.p.c. discret, ou plat.

L'utilisation de fonctions continues va permettre le calcul des dénotations par approximations successives via un calcul de point fixe. Si on reprend l'exemple de la sémantique des énoncés de notre mini-langage impératif du chapitre précédent, l'ensemble Σ des états de la mémoire cède sa place à l'o.p.c. discret avec bottom $\Sigma_\perp$. Sur cet ensemble, on définit la fonction continue $\mathcal{C}[\![c]\!] : \Sigma_\perp \to \Sigma_\perp$. Pour le calcul de $\mathcal{C}[\![\mathit{while}\ b\ \mathit{do}\ c]\!]$, on remplace dans notre raisonnement la fonction partielle par une fonction continue complète en étendant la fonction partielle par les paires $(\sigma, \perp)$ partout où elle n'était pas définie. Le calcul du point fixe consiste alors à définir une fonction initiale à partir des paires (σ, σ) telles que $\mathcal{B}[\![b]\!]$ est fausse, étendue partout ailleurs par $(\sigma, \perp)$ pour les états σ tels que le test est vrai. Ensuite, le calcul de point fixe va «peupler» cette fonction en remplaçant peu à peu les paires $(\sigma, \perp)$ par des paires d'états (σ, σ') obtenues en un tour de boucle, puis deux tours, etc.

5.2 Domaines

La théorie des domaines, c'est-à-dire des ordres partiels complets, est le fondement mathématique de la sémantique dénotationnelle. Avec les fonctions continues, les domaines y joue un rôle central dans la mesure où il assure l'existence de solutions aux équations récursives que l'on est amené à poser dans les définitions des fonctions sémantiques. Ces solutions sont bien entendu obtenues par calcul de points fixes.

De nombreuses constructions bien connues donne naissance à des ordres partiels complets :

1. Tout ensemble ordonné par la relation d'identité est un o.p.c. discret.
2. L'ensemble $\mathcal{P}(X)$ des sous-ensembles d'un ensemble X ordonnée par la relation d'inclusion forme un o.p.c., comme d'ailleurs tout treillis complet.
3. L'ensemble des entiers non-négatifs ω étendu par ∞ ordonné comme une chaîne $0 \sqsubseteq 1 \sqsubseteq \cdots \sqsubseteq n \sqsubseteq \infty$ donne un o.p.c. dénommé Ω.
4. L'ensemble des fonctions partielles $f : X \to Y$ entre deux ensembles X et Y ordonné par l'inclusion forme aussi un o.p.c.

Les fonctions continues sont également courantes dans le paysage methématique. En particulier, toutes les fonctions d'un o.p.c. discret vers un o.p.c. sont continues. Par ailleurs :

1. La fonction identité Id_D sur un o.p.c. D est continue.

2. Soient $f : D \to E$ et $g : E \to F$ des fonctions continues sur des o.p.c. D, E, et F. Leur composition $g \circ f : D \to F$ est continue.

L'utilisation d'o.p.c. et de fonctions continues dans les sémantiques dénotationnelles n'est donc pas une restriction insurmontable.

5.2.1 Motivation

Outre leurs propriétés mathématiques intéressantes dans la résolution des équations récursives, les o.p.c. et les fonctions continues ont une justification plus directe et intuitive dans la sémantique des langages de programmation. Un exemple significatif est celui du calcul sur les séquences, une structure de données que l'on rencontre souvent dans les entrées interactives à des programmes.

Considérons les séquences de symboles 0 et 1 comme une équence de symboles pouvant se terminer par un symbole particulier, ici #. Vu comme des entrées interactives, une séquence peut croître sans borne tant que le symbole de fin # n'a pas été rencontré. Une séquence peut aussi demeurer finie sans être terminée si par exemple le dispositif d'entrée ou une connexion tombe en panne, ou encore si un processus fournissant les données d'entrée entre se met à boucler indéfiniment.

Un ordre partiel intuitif existe sur les séquences : une séquence s est inférieure à une séquence s' si s est un préfixe de s'. L'élément bottom est alors la séquence vide, notée ϵ. Il y a des séquences maximales, 0101#, et des séquences infinies $0000\ldots$, notée 0^ω. Appelons cet o.p.c. S et essayons de définir la fonction $isone : S \to \{\mathbf{true}, \mathbf{false}\}$ qui retourne vrai si la séquence d'entrée contient un 1 et faux sinon. Clairement, $isone(0000\#)$ doit retourner faux, mais que dire de $isone(000)$? On voudrait que le résultat ne soit pas faux car la séquence n'est pas terminée, mais il ne peut être vrai car aucun 1 n'apparaît dans la séquence. On aimerait disposer d'une valeur $\perp$ interprétée comme «ne sait pas», puisque la séquence peut encore croître.

L'o.p.c. discret avec bottom $\mathbf{T}$ est construit sur la base de l'ensemble des valeurs de vérité auquel on adjoint l'élément $\perp$ et que l'on munit de l'ordre $\sqsubseteq$ tel que $\perp \sqsubseteq \perp, \perp \sqsubseteq \mathbf{true}, \perp \sqsubseteq \mathbf{false}, \mathbf{true} \sqsubseteq \mathbf{true}, \mathbf{false} \sqsubseteq \mathbf{true}$. On définit alors la fonction $isone : S \to \mathbf{T}$ telle que :

$$\begin{aligned} isone(1s) &= \mathbf{true} \\ isone(0s) &= isone(s) \\ isone(\#) &= \mathbf{false} \\ isone(\epsilon) &= \perp \end{aligned}$$

Mais même cette définition n'assure pas la continuité de la fonction $isone$. Comment traiter le cas de 0^ω ? On serait tenter de retouner faux, mais la fonction ne serait pas calculable puisque pour s'assurer que l'entrée est bien 0^ω, il faudrait examiner une infinité de symboles. On choisit plutôt $isone(0^\omega) = \perp$. Ainsi, puisque $0^\omega = \bigsqcup_{n\in\omega} 0^n$, et puisque $\forall n.isone(0^n) = \perp$ la fonction $isone$ ne sera continue que si on choisit :

$$isone(0^\omega) = isone(\bigsqcup_{n\in\omega} 0^n) = \bigsqcup_{n\in\omega} isone(0^n) = \perp$$

La continuité de la fonction $isone$ est donc assurée par l'utilisation judicieuse d'o.p.c. comme domaine et codomaine, et il devient possible de l'utiliser dans une expression faisant l'objet d'un calcul de point fixe.

5.2.2 Domaines primitifs et domaines construits

Il existe une grande variété d'ordres partiels complets ou domaines parmi lesquels puiser pour construire les dénotations des phrases dans les sémantiques des langages de programmation. Tout ensemble peut servir à la construction d'un domaine discret, comme nous l'avons vu. De façon générale, de nouveaux domaines peuvent être développés *ex nihilo*, en définissant une relation d'ordre appropriée sur un ensemble d'éléments. Mais l'expérience montre par ailleurs que les domaines couramment utilisés dans les sémantiques présentent une grande régularité dans leur construction.

Ces constructions représentent les constructions courantes sur les données que sont les produits cartésiens, les sommes disjointes (l'union de C) ou encore les espaces de fonctions. Cette observation a donnée naissance à une espèce d'«algèbre» des domaines avec des opérateurs dont la sémantique est de construire de nouveaux domaines à partir de domaines précédemment définis. Dans la plupart des sémantiques utilisant des concepts computationnels courants, cette «algèbre» des domaines suffit à construire tous les domaines nécessaires pour en construire la sémantique. Nous voyons maintenant les principaux domaines de base ainsi que les principales constructions de cette «algèbre» des domaines.

Domaines de base

- $\mathbf{I}$ est le domaine singleton contenant l'unique élément $\bot_{\mathbf{I}}$.
- $\mathbf{O}$ est le domaine à deux valeurs contenant les éléments $\bot_{\mathbf{O}}$ et $\top$.

Valeurs de vérité

- $\mathbf{T}$ est le domaine à trois valeurs $\bot$ **true**, et **false**, muni de l'ordre partiel : $\bot \sqsubseteq \bot, \bot \sqsubseteq$ **true**, $\bot \sqsubseteq$ **false**, **true** $\sqsubseteq$ **true**, **false** $\sqsubseteq$ **true**.
- Tout domaine discret d est muni d'un prédicat d'égalité $e_1 =_d e_2$ qui retourne $\bot_{\mathbf{T}}$ si e_1 ou e_2 est $\bot_d$, sinon elle retourne **true** ou **false**. Ce prédicat est une fonction monotone et continue ssi d est un domaine discret.
- **if** e_1 **then** e_2 **else** e_3 **endif** requiert que e_1 soit une valeur du domaine $\mathbf{T}$ et que e_2 et e_3 soient des valeurs d'un même domaine d. Elle dénote e_2 si $e_1 =$ **true**, e_3 si $e_1 =$ **false** et $\bot_d$ si $e_1 = \bot_{\mathbf{T}}$.

Nombres naturels

- $\mathbf{N}_\bot$ est le domaine discret de nombres naturels auxquels est adjoint un élément $\bot_{\mathbf{N}}$.
- Les fonctions usuelles sont étendues de façon stricte[1] dans tous leurs arguments à $\mathbf{N}_\bot$.

Domaines de fonctions

- $d_1 \to d_2$ dénote le domaine des fonctions continues du domaine dénoté par d_1 vers le domaine dénoté par d_2. On a $f \sqsubseteq_{d_1 \to d_2} g$ ssi $\forall x \in d_1 . f(x) \sqsubseteq_{d_2} g(x)$.
- $\lambda\, x \in d.e$ dénote la fonction continue f définie en prenant $f(x) = e$ où x provient du domaine d.
- $(e_1\ e_2)$ dénote le résultat de l'application de la fonction $f : d \to d'$ dénotée par e_1 sur la valeur $x \in d$ dénotée par e_2.
- $\mathbf{id}_d$ dénote la fonction identité sur le domaine d.
- $e_1 \circ e_2$ dénote la composition des fonctions $f_1 : d \to d'$ et $f_2 : d' \to d''$ dénotées respectivement par e_1 et e_2, telle que $\forall x \in d . e_1 \circ e_2(x) = e_1(e_2(x))$.
- $\mathbf{fix}_d$ dénote l'opérateur de point fixe pour le domaine d; cet opérateur appliqué à une fonction $f : d \to d$ retourne la plus petite solution x telle que $x = f(x)$.
- $d_1 \circ\!\!\to d_2$ dénote la restriction de $d_1 \to d_2$ aux fonctions strictes.
- $\mathbf{strict}_d$ dénote la fonction qui prend une fonction dans $d_1 \to d_2$ et retourne la fonction stricte correspondante dans d, où $d = d_1 \circ\!\!\to d_2$.

[1] Une fonction stricte est une fonction qui évalue tous ses arguments. En terme de domaine, cela veut dire que si l'un des arguments est $\bot$ alors son résultat doit aussi être $\bot$.

$\langle \dots \rangle$	construction de liste
$l \downarrow k$	kth membre de la liste l (partant de 1)
$\#l$	longueur de la liste l
$l \S t$	concaténation des listes l et t
$l \dagger k$	retirer les k premiers membres de la liste l
$x \mid_{\mathrm{D}}$	projection de la valeur x d'un domaine somme quelconque dans sa composante **D**
$i\mathrm{n}\mathbf{D}(x)$	injection de la valeur x dans le domaine somme **D**
$o\mathrm{n}_i(x)$	projection de la valeur x d'un domaine produit quelconque dans son $i$ème composante

FIG. 5.1 – Résumé de la notation utilisée

Domaines produits

- $d_1 \times d_2 \dots \times d_n$ dénote le produit cartésien des domaines d_1, d_2, ..., d_n, pour tout $n \geq 2$. On a $(x_1, \dots, x_n) \sqsubseteq_{d_1 \times d_2 \dots \times d_n} (y_1, \dots, y_n)$ ssi $x_i \sqsubseteq_{d_i} y_i$ pour $i = 1, \dots, n$.
- $(e_1, \dots, e_n)$ représente le n-tuple formé des composants e_1, ..., e_n.
- $\mathbf{on}_i^d$ dénote la fonction de projection $d \to d_i$ d'une valeur du domaine produit d vers son i$^{\mathrm{e}}$ composante (on omet de mentionner le domaine somme d lorsque le contexte ne risque pas d'engendrer de confusion).
- $d_1 \otimes d_2 \dots \otimes d_n$ dénote le produit cartésien discrétisant telle que tout n-tuple contenant une valeur $\perp_{d_i}$ est identifié à $\perp_{d_1 \otimes d_2 \dots \otimes d_n}$. Ainsi, le produit cartésien discrétisant de domaines discrets est lui-même discret.
- $\mathbf{on}_d$ dénote la fonction qui prend une valeur d'un domaine produit $d_1 \times d_2 \dots \times d_n$ et qui retourne la valeur correspondante du domaine produit discrétisant $d_1 \otimes d_2 \dots \otimes d_n$.

Domaines sommes

- $d_1 + d_2 \dots + d_n$ dénote la sommes séparée de domaines d comprenant toutes les valeurs (distinguables) des domaines de la somme, auxquelles est adjoint un nouvel élément $\perp_{d_1 + d_2 \dots + d_n}$. Les éléments de d provenant des différents domaines d_i sont incomparables dans d.
- $d_1 \oplus d_2 \dots \oplus d_n$ dénote la somme unifiée des domaines où toutes les valeurs $\perp_{d_i}$ sont unifiées à la valeur $\perp_{d_1 \oplus d_2 \dots \oplus d_n}$.
- $\mathbf{in}_i^d$ dénote la fonction d'injection d'une valeur d'un domaine d_i dans le domaine somme $d = d_1 + d_2 \dots + d_n$ (ou $d_1 \oplus d_2 \dots \oplus d_n$).
- $x \mid_{d_i}$ dénote la projection d'une valeur $x \in d_1 + d_2 \dots + d_n$ dans la valeur x_i telle que $x = \mathbf{in}_i^d(x_i)$.

Domaines listes

- d^* dénote le domaine des listes de longueur finie composées d'éléménts dans le domaine d sauf $\perp_d$. Les listes de longueurs différentes sont incomparables par $\sqsubseteq$.

La figure 5.1 résume les principales notations utilisées dans les sémantiques dénotationnelles des prochains chapitre pour manipuler les valeurs de domaines courants.

5.3 Un modèle pour le λ-calcul

Le modèle du λ-calcul le plus simple est le modèle P_ω dû à Plotkin. Rappelons qu'un modèle associe un objet mathématique à chaque terme dans la théorie. Pour établir cette

correspondance, la construction suit de près la syntaxe des λ-termes. On cherche une fonction sémantique qui respecte :

$$\begin{aligned} [\![x]\!]\rho &= \rho(x) \\ [\![MN]\!]\rho &= ([\![M]\!]\rho)([\![N]\!]\rho) \\ [\![\lambda x\,.\,M]\!]\rho &= d \in D \mapsto [\![M]\!](\rho;(x,d)) \end{aligned}$$

où ρ représente un environnement de liaison des variables et $d \in D \mapsto \bullet$ représente une fonction prenant un argument d de type D. La notation $\rho(x)$ désigne la valeur associée à la variable x dans ρ et $\rho;(x,d)$ représente l'environnement ρ étendu par la liaison de x à la valeur d.

Tout modèle du λ-calcul cherche à proposer un domaine D dont les éléments constitueront les dénotations des λ-termes. La principale difficulté est l'auto-application des termes sur eux-mêmes. Cette auto-application force à faire en sorte qu'une fonction f représentée par un élément de D ait en même temps pour type $D \to D$. On doit donc trouver un domaine D tel que $D \cong D \to D$.

Voici la contruction du modèle P_ω de Plotkin. Soit P_ω l'ensemble des parties de $\mathbb{N}$ ordonnées par l'inclusion. On obtient un ordre complet, donc *a fortiori* un ordre partiel complet. Une fonction $f : P_\omega \to P_\omega$ est continue si et seulement si :

$$f(x) = \bigcup\{f(a) \mid a\ fini \subseteq x\}$$

Cette propriété est intéressante car elle implique que $n \in f(x)$ ssi il existe une partie finie a de x telle que $n \in f(a)$. On peut donc représenter f par l'ensemble de ces couples (a,n) tels que $n \in f(a)$. L'intérêt de cette propriété est donc que f étant représentée par des ensembles d'objets finis, il est possible d'encoder ces objets sous forme d'entiers, et donc représenter f par des éléments de P_ω. Une partie finie $a = \{i_1, ..., i_n\}$ est encodée par l'entier $m = \sum_{k=1}^{n} 2^{i_k}$. Le décodage de m, noté e_m, reconstruit a en faisant la liste des positions des '1' dans la représentation binaire de m. On dispose par ailleurs d'un codage $\langle\rangle : \mathbb{N}^2 \to \mathbb{N}$. Le couple $(\{i_1, ..., i_n\}, n)$ est donc codé par (m,n), qui lui-même est codé par l'entier $\langle m,n\rangle$; ce double codage est bijectif.

Une fonction continue $f : P_\omega \to P_\omega$ peut donc être encodée par un ensemble d'entiers, et donc un élement de P_ω. Soit :

$$G(f) = \{\langle m,n\rangle \mid n \in f(e_m)\}$$

L'interprétation de l'application ab, pour $a, b \in P_\omega$, consiste alors à rechercher dans a l'entier représentant le couple dont le premier élément est inclut dans b et à retourner le second élément de ce couple :

$$ab = \{n \in \mathbb{N} | (\exists e_m \subseteq b)\langle m,n\rangle \in a\}$$

Muni de ces définitions, on peut maintenant compléter la fonction sémantique donnant la correspondance entre λ-termes et éléments de P_ω :

$$\begin{aligned} [\![x]\!]\rho &= \rho(x) \\ [\![MN]\!]\rho &= ([\![M]\!]\rho)([\![N]\!]\rho) \\ &= \{n \in \mathbb{N} | (\exists e_m \subseteq [\![N]\!]\rho)\langle m,n\rangle \in [\![M]\!]\rho\} \\ [\![\lambda x\,.\,M]\!]\rho &= G(a \in P_\omega \mapsto [\![M]\!](\rho;(x,a))) \\ &= G(\{i_1, ..., i_n\} \in P_\omega \mapsto [\![M]\!](\rho;(x,\{i_1, ..., i_n\}))) \\ &= \{\langle m,n\rangle \mid m = \sum_{k=1}^{n} 2^{i_k} \wedge n \in [\![M]\!](\rho;(x,e_m))\} \end{aligned}$$

Chapitre 6

Outils pour la construction de sémantiques dénotationnelles

Depuis plus de vingt ans, l'approche dénotationnelle a été appliquée à la définition de la sémantique de nombreux langages de programmation, que ce soit des langages ipératifs, fonctionnels ou à objets. Ces diverses applications ont permis d'acquérir une pratique de cette approche et de développer des techniques devenues maintenant usuelles pour définir la sémantique des différentes constructions syntaxiques qui se répètent d'un langage à un autre. Certes, rien n'interdit de développer ses propres techniques. Mais, comme dans d'autres domaines, la sémantique formmelle vise l'échange d'idées entre concepteurs et implanteurs de langages. L'utilisation d'idomes connus rend la communication plus efficace. Dans ce chapitre, nous allons donc voir quelques unes des techniques les plus connues dans la définition de la sémantique dénotationnelle des langages impératifs.

6.1 Un langage impératif

Notre étuede va porter sur un langage impératif de facture très classique dont la syntaxe abstraite est donnée à la figure 6.1. Ce langage propose trois types de données primitifs : les nombres, les caractèress et les booléens. Syntaxiquement, nous ne détaillons pas la forme des nombres et des caractères. Les constantes booléennes vrai et faux ont pour syntaxe `true` et `false` respectivement.

Les expressions de base de ce langage impératif sont formées soit d'un littéral, soit d'une variable, soit de l'application d'un opérateur monadique (*om*) à une sous-expression, soit de l'application d'un opérateur dyadique (*od*) à deux sous-expressions. Les opérateurs monadiques et dyadiques comprennent les opérateurs artihmétiques et booléens classiques. Nous n'avons défini qu'un seul opérateur de comparaison, l'égalité ; les autres opérateurs de comparaison (inégalités) se traitent de manière similaire à ce dernier.

Cinq expressions particulières viennent enrichir ces expressions de base. L'expression de choix (`choix` *e e e*) est une expression d'alternative à la Scheme, qui aura pour résultat le résultat de sa partie alors si le test s'évalue à vrai et celui de sa partie sinon autrement. Cette expression est analogue au `if` de Scheme, mais aussi à l'opérateur '`_? _ : _`' de C et de Java. L'expression (`soit` *dc* `dans` *e*) permet d'introduire des déclarations de constantes locales à des sous-expressions, éventuellement récursive, de manière analogue au `letrec` de Scheme. Une déclaration de constantes (*dc*) se présente comme une séquence de déclarations individuelles

Domaines syntaxiques :

n	$\in$	Num
car	$\in$	Caractères
k	$\in$	Littéraux
ν	$\in$	Identificateurs
om	$\in$	OpérateursMonadiques
od	$\in$	OperateursDyadiques
e	$\in$	Expressions
dc	$\in$	DeclarationConstantes
dp	$\in$	DeclarationParametre
$decv$	$\in$	DeclarationVariables
t	$\in$	Type
c	$\in$	Commandes
p	$\in$	Programmes

k	::=	`true` | `false` | n | car
om	::=	`-` | $\neg$
od	::=	$+$ | $-$ | $\times$ | $\div$ | $\wedge$ | $\vee$ | $=$
e	::=	k | ν | $om\ e$ | $e\ od\ e$ | `(choix` $e\ e\ e$`)` | `(soit` dc `dans` e`)` | `(fun (`dp`)` e`)` | e `(` e `)` | `(proc (`dp`)` c`)`
dc	::=	`val` $\nu\ e$ | dc `;` dc | `rec` dc
dp	::=	`val` $\nu\ t$ | `var` $\nu\ t$
$decv$	::=	`var` $\nu\ t$ | $decv$ `,` $decv$
t	::=	`bool` | `num`
c	::=	`skip` | e `:=` e | c `;` c | `alternative` $e\ c\ c$ | `tantque` $e\ c$ | `begin` dv `;` c `end` | e `(` e `)` | `lire` e | `ecrire` e
p	::=	`prog` $dc\ c$

FIG. 6.1 – Domaines syntaxiques le mini-langage imperatif

séparées par des points-virgules. Une déclaration individuelle est soit la déclaration d'une valeur introduite par le mot-clé *val* et suivi d'un nom de variable et d'un expression, soit une déclaration récursive introduite par le mot-clé *rec* suivie d'un séquence de déclarations.

Les formes syntaxique `(fun (`dp`)` e`)` et `(proc (`dp`)` e`)` permettent de créer des fonctions et des procédures. Dans les deux cas, une déclaration de paramètres permet de définir le paramètre de la fonction ou de la procédure. Sans perte de généralité, nous nous limitons donc à des fonctions à un seul argument. Un paramètre introduit par le mot-clé `val` est un paramètre en entrée passé par valeur alors que celui introduit par le mot-clé `var` est un paramètre en entrée/sortie passé par référence. La forme syntaxique e `(` e `)` est l'application de la fonction résultant de la première sous-expression au résultat de la seconde sous-expression. Une déclaration de paramètre fait aussi apparaître un type.

La déclaration de variables est une liste de déclarations individuelles, chacune introduite par le mot-clé *var* suivi d'un nom de variable et de son type.

Les énoncés (aussi appelés traditionnellement commandes) comportent la commande `skip`,

qui ne fait rien, l'affectation, et la séquence, c'est-à-dire une suite d'énoncés séparés par des points-virgules. L'énoncé d'alternative, de forme `alternative` e c c est formée d'une expression booléenne et de deux commandes en partie alors et sinon. L'énoncé de répétition, de forme `tantque` e c, comprte une expression booléenne contrôlant la répétition et le corps de la répétition qui est un énoncé. Un bloc, de la forme `begin` dv ; c `end`, est délimité par les mots-clés `begin` et `end` et il est constitué de délcarations de variables suivies d'un énoncé. L'appel de procédure prend une forme similaire à l'appel de fonction. Les deux derniers énoncés sont des énoncés d'entrée/sortie. L'énoncé `lire` e lit une valeur et la range dans la variable passée en paramètre. L'énoncé `ecrire` e écrit le résultat de son argument.

Un programme est introduit par le mot-clé `prog` et il est constitué d'une série de déclarations de constantes et d'un énoncé évaluer dans le contexte de ces déclarations.

6.2 Choix des domaines sémantiques

La définition des domaines sémantiques apparaît à la figure 6.3. Cette définition est intimement lié au choix de mise en œuvre des fonctions de valuations. Certains des choix ne seront donc complètement justifiées que par la suite.

Les domaines pour les littéraux sont pour la plupart non-spécifiés.[1] Pour les besoins du reste de la sémantique, on fait tout de même apparaître le domaine **T** des valeurs de vérités que nous avons présenté comme un domaine standard au chapitre précédent. Le domaine **V**, pour *values*, réunit sous la forme d'une somme de domaines toutes les valeurs des constantes du langage.

La définition des domaines sémantiques pour un langage de programmation s'organise autour de quelques domaines, appelées *domaines sémantiques caractéristiques*, qui jouent un rôle crucial dans la définition de la sémantique du langage. Ces domaines sont également les premiers que vont analyser les concepteurs et implanteurs de langages pour se faire une idée de la puissance d'expression d'un nouveau langage. De façon à simplifier l'analyse des sémantiques, les sémanticiens observent donc une règle de nommage de ces domaines qui en facilite le repérage rapide.

Le premier domaine caractéristique pour un langage est le domaine **V**, pour *values*, qui indique l'ensemble des constantes que l'on peut exprimer syntaxiquement dans un programme. Dans notre langage impératif, les constantes exprimables directement sont celles de type numérique (sans plus de détails sur le fait que ce soient des entiers relatifs, des réels, etc.), des booléens et des caractères. Il s'agit là de constantes assez standard.

Le second domaine caratéristique est le domaine **EV**, pour *expression values*, qui indique l'ensemble des valeurs qui peuvent résulter de l'évaluation d'une expression. Généralement toutes les valeurs constantes du langage, c'est-à-dire **V**, font partie de **EV**. Une variable étant une expression, toute valeur pouvant être rangée dans une variable est aussi dans **EV**. L'application de fonction étant aussi une expression, toute valeur retournable comme résultat d'une fonction doit aussi faire partie de **EV**. Enfin, notre langage prévoit des expressions pour créer des fonctions et des procédures ; les valeurs exprimant fonctions et procédures font donc aussi partie de **EV**.

Parmi les valeurs de **EV**, on distingue généralement deux sous-domaines : les domaines **LV**, pour *left-values*, et **RV**, pour *right-values*. Cette distinction est introduite par l'énoncé

[1] Dans ce cas, il s'agit d'une pratique courante qui évite de cacher l'essentiel de la sémantique sous une masse de détails simplistes.

V : pour *values*, c'est-à-dire les valeurs qui peuvent être directement exprimées dans le langage (littéraux).

EV : pour *expression values*, c'est-à-dire les valeurs qui peuvent résulter de l'évaluation d'une expression.

DV : pour *denotable values*, c'est-à-dire les valeurs qui peuvent être désignées par un identificateur (constantes, fonctions, types, etc.).

PV : pour *parameter values*, c'est-à-dire les valeurs qui peuvent être passées en paramètres à une fonction ou une procédure.

FV : pour *function values*, c'est-à-dire les valeurs qui peuvent être retournées comme résultat d'une fonction.

SV : pour *storable values*, c'est-à-dire les valeurs qui peuvent être rangées dans un emplacement mémoire.

LV : pour *left-values*, c'est-à-dire les valeurs pouvant apparaître à gauche d'une affectation.

RV : pour *right-values*, c'est-à-dire les valeurs pouvant apparaître à droite d'une affectation (dont nécessairement **SV**).

FIG. 6.2 – Domaines sémantiques caractéristiques

d'affectation, les valeurs à gauche étant celles utilisées à gauche de l'affectation alors que les valeurs à droites sont celles utilisables à droite de l'affectation. Dans notre syntaxe, nous avons permis d'utiliser une expression à gauche d'une affectation. Cela permet une plus grande flexibilité dans l'utilisation d'une affectation[2] ou encore nous permettrait d'introduire facilement les tableaux et donc l'indiçage en partie gauche de l'affectation.

Tout langage de programmation donne la possibilité de désigner certaines entités par des identificateurs. Bien entendu, cela inclut les variables, qui seront traitées plus particulièrement ensuite, mais plus généralement la désignation de fonctions et procédures par des noms, voire des constantes ou des types. Le troisième domaine sémantique caractéristique est celui des valeurs que l'on peut désigner ainsi par un identificateur dans le langage est **DV**, pour *denotable values*. Dans notre langage, les valeurs de **DV** sont les mêmes que celles dans **EV**.

Lorsque le langage permet de définir des fonctions et des procédures, l'appel de fonction ou de procédure nécessite le passage de paramètres réels. Le quatrième domaine sémantique caractéristique est **PV**, pour *parameter values*, des valeurs que l'on peut passer en paramètres aux fonctions et procédures. Les questions intéressantes que l'on se pose généralement en examinant **PV** est de savoir s'il permet le passage par référence, et s'il permet le passage de fonctions ou de procédures.

Lorsque le langage permet la définition de fonctions ou de procédures fonctionnelles, le domaine **FV**, pour *function values*, est un domaine sémantique caractéristique indiquant les valeurs pouvant être retournées comme résultat d'une fonction.

Pour gérer les variables, on utilise la mémoire, appelée en anglais *store*. Le domaine sémantique caractéristique qui définit les valeurs que l'on peut ranger dans les variables est le domaine **SV**, pour *storable values*. On limite le domaine **SV** aux valeurs que peut contenir un emplacement mémoire. Dans le cas de notre langage, cela inclut les nombres, les valeurs de

[2]Cela permet d'écrire des choses comme '`choix (x = y) x y = 2 * y`' qui affectera la valeur de '`2 * y`' à la variable `x` ou `y` selon que les deux variables sont égales ou non.

Domaines sémantiques :

Domaines pour les littéraux et les domaines sémantiques caractéristiques :

$$
\begin{array}{rcll}
\tau & \in & \mathbf{T} & = \{\bot_{\mathbf{T}}, \mathbf{true}, \mathbf{false}\} \\
n & \in & \mathbf{Num} & = \textit{non-spécifié} \\
ch & \in & \mathbf{Char} & = \textit{non-spécifié} \\
\upsilon & \in & \mathbf{V} & = \mathbf{T} \oplus \mathbf{Num} \oplus \mathbf{Char} \\
\omega & \in & \mathbf{SV} & = \mathbf{T} \oplus \mathbf{Num} \oplus \mathbf{Char} \\
\texttt{in} & \in & \mathbf{In} & = \mathbf{SV}^* \\
\texttt{out} & \in & \mathbf{Out} & = \mathbf{SV}^* \\
 & & \mathbf{FV} & = \mathbf{V} \oplus \mathbf{F} \oplus \mathbf{P} \\
f & \in & \mathbf{F} & = \mathbf{S} \to \mathbf{PV} \to \mathbf{FV} \\
proc & \in & \mathbf{P} & = \mathbf{PV} \to (\mathbf{In} \otimes \mathbf{Out} \otimes \mathbf{S}) \to (\mathbf{In} \otimes \mathbf{Out} \otimes \mathbf{S}) \\
\iota & \in & \mathbf{LV} & = \mathbf{Loc} \\
\varepsilon & \in & \mathbf{EV} & = \mathbf{V} \oplus \mathbf{F} \oplus \mathbf{LV} \oplus \mathbf{P} \\
\delta & \in & \mathbf{DV} & = \mathbf{V} \oplus \mathbf{F} \oplus \mathbf{LV} \oplus \mathbf{P} \\
\vartheta & \in & \mathbf{RV} & = \mathbf{V} \oplus \mathbf{F} \oplus \mathbf{P} \\
\pi & \in & \mathbf{PV} & = \mathbf{V} \oplus \mathbf{F} \oplus \mathbf{LV} \oplus \mathbf{P}
\end{array}
$$

Domaines sémantiques pour les environnements et les mémoires :

$$
\begin{array}{rcll}
t & \in & \mathbf{O} & = \{\bot_{\mathbf{O}}, \top\} \\
\nu & \in & \mathbf{Ide} & = \textit{non-spécifié} \\
p & \in & \mathbf{Pointer} & = \textit{non-spécifié} \\
l & \in & \mathbf{Loc} & = \mathbf{Pointer} \oplus \mathbf{O} \\
\rho & \in & \mathbf{Env} & = \mathbf{Ide} \to (\mathbf{DV} \oplus \mathbf{O}) \\
\sigma & \in & \mathbf{S} & = \mathbf{Loc} \to (\mathbf{SV} \oplus \mathbf{T})
\end{array}
$$

FIG. 6.3 – Domaines sémantiques pour notre langage imperatif

vérité et les caractères.

La figure 6.2 résume la définition des domaines sémantiques caractéristiques, tandis que la figure 6.3 donne leur contenu pour notre langage impératif simple. Cette seconde figure donne aussi le contenu d'autres domaines sémantiques importants :

- **In** et **Out** qui sont les domaines des valeurs passées en entrée et obtenues en sortie du programme, c'est-à-dire le fichier d'entrée standard et le fichier de sortie standard,
- **F** qui est le domaine des valeurs représentant les fonctions, de même que
- **P** qui est le domaine des valeurs représentant les procédures.

Dans le cas de **F**, on voit que la fonction prend l'état mémoire (**S**) en paramètre mais ne le retourne pas en résultat ; cela vient du fait que l'état sert à obtenir les valeurs des variables, mais que la fonction ne peut pas faire de modifications à cet état. Les procédures sont représentées par des fonctions prenant en paramètre la mémoire, le fichier d'entrée et le fichier de sortie et retourne un nouvel état mémoire, un nouvel état du fichier d'entrée et un nouvel état du fichier de sortie. La procédure peut donc faire des affectations, lire une valeur et écrire une valeur.

6.3 Identificateurs, environnements et mémoires

La plupart des langages de programmation utilisent aujourd'hui une notion de portée des variables dite statique. La portée d'une variable est par définition la portion du programme dans laquelle une référence à une variable donnée est possible. La notion de portée statique a

été introduite au chapitre 2 dans le contexte de Scheme.

Pour donner une sémantique à cette notion de portée statique, les sémanticiens utilise un technique qui découple la liaison d'un nom de variable à une adresse mémoire, une notion. Cette liaison est statique puisqu'elle dépend essentiellement du texte de programme et non de l'exécution. La liaison de l'adresse mémoire au contenu du mot mémoire correspondant est par contre une notion dynamique puisque l'état de la mémoire change au cours de l'exécution d'un programme. Lorsqu'une référence à une variable apparaît dans un programme, retrouver la valeur correspondante est réalisé par un mécanisme en deux temps : retrouver l'adresse mémoire à laquelle est liée la variable dans le contexte courant, puis retrouver le contenu associé à l'adresse mémoire identifiée.

La liaison entre variables et adresses mémoire est traditionnellement réalisé par un environnement. Un environnement est une fonction qui prend un nom de variable et qui retourne l'adresse à laquelle est liée ce nom de variable à l'endroit où la référence à cette variable est faite dans le programme. La liaison entre les adresses mémoire et la valeur contenue dans le mot mémoire correspondant est traditionnellement réalisée par l'intermédiaire d'une mémoire, appelée en anglais *store*. Le mécanisme de référence à double détente décrit ci-haut revient donc à :

$$\mathit{variable} \xrightarrow{\mathit{env}} \mathit{adresse} \xrightarrow{\mathit{mémoire}} \mathit{valeur}$$

Environnements

Les environnements sont donc modélisés par des fonctions prenant un identificateur et retournant la valeur à laquelle cet identificateur est lié. Outre les variables déjà discutées, l'environnement doit plus généralement servir à la liaison de tout identificateur à une valeur dénotable dans **DV**. Cette notion est plus générale que celle de variable dans la mesure où elle doit nous permettre de traiter la définition de constante, introduite dans notre langage par l'expression '**soit ... dans ...**'. Puisque le domaine **DV** inclut les valeurs à gauche (adresses), si un environnement lie un identificateur à une valeur dans **DV**, nous allons pouvoir traiter à la fois la lisiaon des constantes et celle des variables.

Maintenant, une question qui se pose est quel statut peut avoir une variable dans un certain environnement ? On a choisi de distinguer le cas où la variable n'est pas définie dans l'environnement et le cas où la variable est définie multiplement sans qu'une priorité n'ait été donnée à l'une ou l'autre des définitions[3] Lorsqu'une variable est définie multiplement mais que l'une des définition a priorité sur l'autre (comme une variable locale qui masque une variable globale), c'est la définition prioritaire qui sera choisie.

Ainsi, plutôt que de simplement retourner la valeur associée à une variable (dans **DV**), la fonction représentant un environnement pourra aussi retourner l'un des deux marqueurs $\{\bot_{\mathbf{O}}, \top\} \in \mathbf{O}$ où $\top$ dénotera le fait que l'identificateur n'est pas défini dans l'environnement alors que $\bot_{\mathbf{O}}$ dénotera le fait que l'identificateur est multiplement défini. L'environnement ρ est donc une fonction dont le type est :

$$\rho \in \mathbf{Env} = \mathbf{Ide} \to (\mathbf{Loc} \oplus \mathbf{O})$$

L'identificateur *void* désigne l'environnement vide et il est défini par :

$\mathit{void} : : \mathbf{Env}$

$\mathit{void} = \lambda\nu.\mathit{in}(\mathbf{DV} \oplus \mathbf{O})(\top)$

[3]Ce qui sera interdit dans la plupart des cas, mais qui peut mener à des messages d'erreur différents.

Cette fonction retourne toujours $\top$, comme il se doit.

Les fonctions *bound* et *binding* servent respectivement à consulter un environnement pour retrouver la valeur associée à une variable et à étendre un environnement par une nouvelle liaison d'une variable à une valeur dans **DV**. *bound* rtournera toujours une valeur dans **DV** ; cette vameur sera $\perp_{\mathbf{DV}}$ si l'identificateur n'est pas défini ou est défini multiplement sans ordre de priorité :

$bound : : (\mathbf{Ide} \otimes \mathbf{Env}) \rightarrow \mathbf{DV}$

$bound =$
$\quad \lambda(\nu, \rho).$ **case** $tmp = \rho(\nu)$ **of**
$\qquad$ $is\mathbf{O}\ ? \Rightarrow \perp_{\mathbf{DV}}$,
$\qquad$ $is\mathbf{DV}\ ? \Rightarrow tmp\ |_{\mathbf{DV}}$
$\qquad$ **endcase**

La forme **case** utilisée ici permet de veérifier l'appartenance d'une valeur d'un domaine somme à l'une ou l'autre des parties de la somme. Dans le cas présent, le résultat d'un environnement est une valeur dans $\mathbf{DV} \otimes \mathbf{O}$; le **case** permet donc de vérifier si le résultat de $\rho(\nu)$ est dans **DV** ou **O**.

La fonction *binding* prend un identificateur ν, une valeur δ et un environnement ρ, et produit un environnement étendu, c'est-à-dire une fonction qui prend un identificateur ν_1 et qui retourne δ si ν est égal à ν_1 ou $\rho(\nu_1)$ sinon :

$binding : : (\mathbf{Ide} \otimes \mathbf{DV}) \rightarrow \mathbf{Env}$

$binding =$
$\quad \lambda(\nu, \delta).\lambda\nu_1.$ **if** $\nu_1 =_{\mathbf{Ide}} \nu$
$\qquad$ **then** $in(\mathbf{DV} \oplus \mathbf{O})(\delta)$
$\qquad$ **else** $in(\mathbf{DV} \oplus \mathbf{O})(\top)$
$\qquad$ **endif**

Les fonctions *overlay* et *combine* servent toutes les deux à prendre deux environnements et à les combiner pour en produire un seul. La fonction *overlay* réalise cette combinaison en permettant aux liaisons du premier environnement de masquer celle du second environnement. Cette forme de combinaison est utilisée, par exemple, lorsqu'il faut étendre l'environnement courant avec les liaisons des paramètres formels d'une fonction à ses paramètres réels. La fonction *combine* ne permet pas la redéfinition d'une variable ; cette forme de combinaison est utilisée lorsqu'il faut traiter l'ensemble des liaisons introduites par une expression `soit` ou encore par une liste de déclaration de variables dans un énoncé `begin`.

$overlay : : (\mathbf{Env} \otimes \mathbf{Env}) \rightarrow \mathbf{Env}$

$overlay =$
$\quad \lambda(\rho, \rho_1).\lambda\nu.$ **case** $tmp = \rho(\nu)$ **of**
$\qquad$ $is\mathbf{O}\ ? \Rightarrow \rho_1(\nu)$,
$\qquad$ $is\mathbf{DV}\ ? \Rightarrow tmp$
$\qquad$ **endcase**

$combine : : (\mathbf{Env} \otimes \mathbf{Env}) \rightarrow \mathbf{Env}$

$combine =$
$\quad \lambda(\rho, \rho_1).\lambda\nu.$ **case** $tmp = \rho(\nu)$ **of**
$\qquad$ $is\mathbf{O}\ ? \Rightarrow \rho_1(\nu)$,
$\qquad$ $is\mathbf{DV}\ ? \Rightarrow$ **case** $tmp_1 = \rho_1(\nu)$ **of**
$\qquad\qquad$ $is\mathbf{O}\ ? \Rightarrow tmp$,
$\qquad\qquad$ $is\mathbf{DV}\ ? \Rightarrow in(\mathbf{DV} \oplus \mathbf{O})(\perp_{\mathbf{O}})$

endcase
endcase

La mémoire (ou *store*) est aussi modélisée par une fonction prenant une adresse mémoire, du domaine **Loc**, et qui rend normalement une valeur contenue dans l'espace mémoire ainsi adressé, du domaine **SV**. Cependant, comme pour l'environnement, il est utile de diposer de marqueurs pour distinguer un espace mémoire libre, d'un espace mémoire alloué mais non encore initialisé, d'un espace mémoire alloué et contenant une valeur significative. Pour cela, on construit le domaine des valeurs contenues dans la mémoire en prenant la somme des domaines **SV** et **T**. Dans ce cas, les valeurs du domaine **T** vont servir, pour **false**, à dénoter l'espace mémoire libre et pour `true` l'espace mémoire alloué mais non intialisé. La mémoire est donc une fonction dont le type est :

$$\sigma \in \mathbf{Loc} \rightarrow (\mathbf{SV} \oplus \mathbf{T})$$

À partir de cette description, il apparaît clairement que la la mémoire vide est représentée par la fonction qui pour toute valeur dans **Loc** retourne la valeur **false** injectée dans le domaine somme $\mathbf{SV} \oplus \mathbf{T}$:

$$\textit{empty-store} = \lambda l.\textit{in}(\mathbf{SV} \oplus \mathbf{T})(\mathbf{false})$$

Les fonctions *store* et *stored* jouent ici le même rôle que les fonctions *binding* et *bound* pour les environnements : la liaison d'une adresse à une nouvelle valeur et la consultation d'un espace mémoire à partir d'une adresse.

$$\begin{array}{l}\textit{store} = \\ \quad \lambda(l,\omega,\sigma).\lambda l_1.\ \ \textbf{if}\ l =_{\mathbf{Loc}} l_1 \\ \qquad\qquad\qquad\ \textbf{then}\ \textit{in}(\mathbf{SV} \oplus \mathbf{T})(\omega) \\ \qquad\qquad\qquad\ \textbf{else}\ \sigma(l_1) \\ \qquad\qquad\qquad\ \textbf{endif}\end{array}$$

$$\begin{array}{l}\textit{stored} = \\ \quad \lambda(l,\sigma).\ \ \textbf{let}\ \textit{tmp} = \sigma(l) \\ \qquad\qquad\ \textbf{in}\ \ \textbf{if}\ \textit{is}\mathbf{T}\ ?(\textit{tmp}) \\ \qquad\qquad\qquad\ \textbf{then}\ \perp_{\mathbf{SV}} \\ \qquad\qquad\qquad\ \textbf{else}\ \textit{tmp}\ |_{\mathbf{SV}} \\ \qquad\qquad\qquad\ \textbf{endif}\end{array}$$

La fonction *stored* retourne la valeur $\perp_{\mathbf{SV}}$ dès que l'emplacement mémoire consulté est libre ou non initialisé.

La gestion de la mémoire est réalisée à l'aide des quatre fonctions *location*, *reservation*, *allocation* et *free*. La fonction *location* prend un état mémoire et retourne l'adresse d'un espace mémoire libre. Cette fonction n'est pas spécifiée, laissant libre l'implanteur de la réaliser au mieux sur le matériel choisi pour implanter le langage :

$$\textit{location}(\sigma) = \textit{unspecified}$$

La fonction *reservation* prend une adresse et un état mémoire dans lequel cette adresse désigne un espace mémoire libre, et retourne un nouvel état mémoire dans lequel l'espace désigné par l'adresse est marqué comme alloué mais non-initialisé :

$$\textit{reservation} =$$

$\lambda(l,\sigma).\lambda l_1.$ **if** $l =_{\mathbf{Loc}} l_1$
then $in(\mathbf{SV} \oplus \mathbf{T})(\mathbf{true})$
else $\sigma(l_1)$
endif

La fonction *allocation* sert à allouer un espace mémoire pour conternir la valeur d'une variable. Elle consiste à requérir l'adresse d'un espace mémoire libre et à réserver cet espace. Elle retourne l'adresse de l'espace mémoire alloué et l'état mémoire résultant de la réservation :

$allocation =$
$\lambda\sigma.$ **let** $l = location(\sigma)$
in $(l, reservation(l,\sigma))$

Enfin, la fonction *free* sert à libérer un espace mémoire préalablement alloué. Il suffit pour l'adresse donnée de construire un nouvel état mémoire dans lequel l'espace ainsi désigné va maintenant contenir la valeur **false** :

$free =$
$\lambda(l,\sigma).\lambda l_1.$ **if** $l =_{\mathbf{Loc}} l_1$
then $in(\mathbf{SV} \oplus \mathbf{T})(\mathbf{false})$
else $\sigma(l_1)$
endif

6.4 Expressions

Les expressions sont traitées par le fonction sémantique $\mathcal{E}$ dont les équations sémantiques sont regroupées à la figure 6.8. Une expression s'évalue dans un contexte qui comprend un environnement courant et un état mémoire courant. Cette évaluation retourne une valeur dans le domaine **EV** comme il se doit. La signature de $\mathcal{E}$ est donc :

$$\mathcal{E} :: \text{Expressions} \rightarrow (\mathbf{Env} \otimes \mathbf{S}) \rightarrow \mathbf{EV}$$

Considérons chacun des cas dans l'ordre. Le traitement des constantes consiste simplement à trouver la valeur sémantique (dans **V**) correspondant à la valeur exprimée syntaxiquement ($k \in$ Littéraux) ; cette valeur est simplement injectée dans le domaine **EV** :

$\mathcal{E}[\![k]\!] = \lambda(\rho,\sigma).in\mathbf{EV}(\mathcal{L}[\![k]\!])$

Pour les identificateurs, il suffit de consulter l'environnement courant. De deux choses l'une : soit l'identificateur est lié à une constante, une fonction ou une procédure, et alors cette valeur devient immédiatement disponible, soit l'identificateur est celui d'une variable et alors c'est l'adresse de l'emplacement mémoire alloué pour cette variable qui est retournée (toutes valeurs dans **DV** injectée dans **EV**). On ne déréférence pas la variable, dans la mesure où dans certain contexte (gauche de l'affectation, passage par référence, etc.), on peut avoir besoin de cette adresse plutôt que de la valeur de la variable.

$\mathcal{E}[\![\nu]\!] = \lambda(\rho,\sigma).bound(\nu,\rho)$

Le traitement des expressions construites à partir d'opérateurs monadiques et dyadiques utilisent des fonctions définies sur les domaines des litéraux de base **Num** et **T**. Les opérateurs étant des catégories syntaxiques à part, on définit deux fonctions sémantiques $\mathcal{OM}$ et $\mathcal{OD}$ qui

$$\mathcal{OM} :: \text{OpérateursMonadiques} \to \mathbf{EV} \to \mathbf{EV}$$
$$\mathcal{OM}[\![\text{-}]\!] = \lambda\varepsilon.in\mathbf{EV}(in\mathbf{V}(0 - \varepsilon\ |_{\mathbf{V}}|_{\mathbf{Num}}))$$
$$\mathcal{OM}[\![\neg]\!] = \lambda\varepsilon.in\mathbf{EV}(in\mathbf{V}(\neg\varepsilon\ |_{\mathbf{V}}|_{\mathbf{T}}))$$

$$\mathcal{OD} :: \text{OpérateursDyadiques} \to (\mathbf{EV} \otimes \mathbf{EV}) \to \mathbf{EV}$$
$$\mathcal{OD}[\![+]\!] = \lambda(\varepsilon,\varepsilon_1).in\mathbf{EV}(in\mathbf{V}(\varepsilon\ |_{\mathbf{V}}|_{\mathbf{Num}} + \varepsilon_1\ |_{\mathbf{V}}|_{\mathbf{Num}}))$$
$$\mathcal{OD}[\![-]\!] = \lambda(\varepsilon,\varepsilon_1).in\mathbf{EV}(in\mathbf{V}(\varepsilon\ |_{\mathbf{V}}|_{\mathbf{Num}} - \varepsilon_1\ |_{\mathbf{V}}|_{\mathbf{Num}}))$$
$$\mathcal{OD}[\![\times]\!] = \lambda(\varepsilon,\varepsilon_1).in\mathbf{EV}(in\mathbf{V}(\varepsilon\ |_{\mathbf{V}}|_{\mathbf{Num}} \times \varepsilon_1\ |_{\mathbf{V}}|_{\mathbf{Num}}))$$
$$\mathcal{OD}[\![\div]\!] = \lambda(\varepsilon,\varepsilon_1).in\mathbf{EV}(in\mathbf{V}(\varepsilon\ |_{\mathbf{V}}|_{\mathbf{Num}} \div \varepsilon_1\ |_{\mathbf{V}}|_{\mathbf{Num}}))$$
$$\mathcal{OD}[\![\wedge]\!] = \lambda(\varepsilon,\varepsilon_1).in\mathbf{EV}(in\mathbf{V}(\varepsilon\ |_{\mathbf{V}}|_{\mathbf{T}} \wedge \varepsilon_1\ |_{\mathbf{V}}|_{\mathbf{T}}))$$
$$\mathcal{OD}[\![\vee]\!] = \lambda(\varepsilon,\varepsilon_1).in\mathbf{EV}(in\mathbf{V}(\varepsilon\ |_{\mathbf{V}}|_{\mathbf{T}} \vee \varepsilon_1\ |_{\mathbf{V}}|_{\mathbf{T}}))$$
$$\mathcal{OD}[\![=]\!] = \lambda(\varepsilon,\varepsilon_1).in\mathbf{EV}(in\mathbf{V}(\varepsilon = \varepsilon_1))$$

FIG. 6.4 – Fonctions de valuation des opérateurs (monadiques et dyadiques)

vont lier les opérateurs syntaxiques à la fonction qu'ils dénotent. Les équantions sémantiques de ces deux fonctions sémantiques apparaissent à la figure 6.4. Le traitement des expressions avec opérateur monadique consiste donc à trouver la fonction dénotée par l'opérateur syntaxique, à évaluer l'expression sur laquelle l'opérateur est appliqué, et à appliquer la fonction sur le résultat de cette évaluation. Il convient cependant de bien gérer les conversions d'un domaine à l'autre pour respecter les types des fonctions. Pour les opérateurs dyadiques, le précédés est similaire à ceci près qu'il y a deux sous-expressions à évaluer.

$$\mathcal{E}[\![om\ \ e]\!] =$$
$$\lambda(\rho,\sigma).\ \ \textbf{let}\ fnm = \mathcal{OM}[\![om]\!]$$
$$\textbf{and}\ \varepsilon = \mathcal{R}[\![e]\!](\rho,\sigma)$$
$$\textbf{in}\ fnm(\varepsilon)$$

$$\mathcal{E}[\![od\ \ e\ \ e_1]\!] =$$
$$\lambda(\rho,\sigma).\ \ \textbf{let}\ fnd = \mathcal{OD}[\![od]\!]$$
$$\textbf{and}\ \varepsilon = \mathcal{R}[\![e]\!](\rho,\sigma)$$
$$\textbf{and}\ \varepsilon_1 = \mathcal{R}[\![e_1]\!](\rho,\sigma)$$
$$\textbf{in}\ fnd(\varepsilon,\varepsilon_1)$$

L'évaluation des sous-expressions utilisent souvent la fonction sémantique $\mathcal{R}$. Cette fonction sémantique sert à forcer l'obtention d'une valeur à droite lors de l'évaluation d'une expression. Ainsi, à chaque fois qu'une expression est évaluée dans un contexte où elle doit retourner une valeur à droite, cette expression est évaluée en utilisant $\mathcal{R}$. $\mathcal{R}$ se contente alors d'évaluer cette expression en utilisant la fonction $\mathcal{E}$. Si le résultat de l'évaluation est une valeur des domaines **V**, **F** et **P**, cette valeur est directement retournée/. Si par contre le résultat est une valeur à gauche, dans **LV**, cette valeur est déréférencée dans la mémoire courante pour obtenir la valeur contenue dans l'espace mémoire correspondant.

La cas de l'expression d'alternative `choix` est relativement simple. L'expression servant de test de l'alternative est évaluée et doit retourner un résultat dans la partie **T** du domaine somme **EV**. Ce résultat est utilisé dans la fonction **if then else endif** définie sur le domaine

$$\mathcal{DC} :: \text{DeclarationConstante} \rightarrow (\mathbf{Env} \otimes \mathbf{S}) \rightarrow \mathbf{Env}$$

$$\mathcal{DC}[\![\texttt{(val } \nu\ e\texttt{)}]\!] =$$
$$\quad \lambda(\rho,\sigma).\ \mathbf{let}\ \varepsilon = \mathcal{E}[\![e]\!](\rho,\sigma)$$
$$\quad\quad\quad \mathbf{in}\ \mathit{binding}(\nu,\varepsilon)$$

$$\mathcal{DC}[\![dc\ \texttt{;}\ dc_1]\!] =$$
$$\quad \lambda(\rho,\sigma).\ \mathbf{let}\ \rho_1 = \mathcal{DC}[\![dc]\!](\rho,\sigma)$$
$$\quad\quad\quad \mathbf{and}\ \rho_2 = \mathcal{DC}[\![dc_1]\!](\rho,\sigma)$$
$$\quad\quad\quad \mathbf{in}\ \mathit{combine}(\rho_1,\rho_2)$$

$$\mathcal{DC}[\![\texttt{(rec } dc\texttt{)}]\!] = \lambda(\rho,\sigma).\mathbf{fix}(\lambda\rho_1.\mathcal{DC}[\![dc]\!](\mathit{overlay}(\rho_1,\rho),\sigma))$$

FIG. 6.5 – Fonction de valuation des déclarations de constantes

T et qui évalue l'une ou l'autre de ses parties alors ou sinon selon la valeur vrai ou faux de son premier argument.

$$\mathcal{E}[\![\texttt{(choix } e\ \ e_1\ \ e_2\texttt{)}]\!] =$$
$$\quad \lambda(\rho,\sigma).\ \mathbf{let}\ \varepsilon = \mathcal{R}[\![e]\!](\rho,\sigma)$$
$$\quad\quad\quad \mathbf{in}\ \ \mathbf{if}\ \varepsilon\ |_{\mathbf{V}}|_{\mathbf{T}}$$
$$\quad\quad\quad\quad\quad \mathbf{then}\ \mathcal{E}[\![e_1]\!](\rho,\sigma)$$
$$\quad\quad\quad\quad\quad \mathbf{else}\ \mathcal{E}[\![e_2]\!](\rho,\sigma)$$
$$\quad\quad\quad\quad\quad \mathbf{endif}$$

L'expression `soit` permettant la liaison d'identificateurs fait intervenir deux sous-expressions : la liste des liaisons et le corps qui est évalué dans le contexte étendu par les liaisons. Pour le traitement des liaisons, on fait appel à la fonction sémantique $\mathcal{DC}$ qui est définie à la figure 6.5. Le traitement des liaisons demande à évaluer les expressions donnant les valeurs à lier aux identificateurs, puis à étendre l'environnement avec les liaisons des identificateurs à ces valeurs. Pour l'évaluation des expresions, il faut connaître l'environnement et la mémoire courants. Puisqu'il ne s'agit que de lier des identificateurs à des valeurs, il est raisonnable de retourner comme résultat un environnement représentant ces liaisons. La signature de la fonction sémantique $\mathcal{DC}$ est donc :

$$\mathcal{DC} :: \text{DeclarationConstante} \rightarrow (\mathbf{Env} \otimes \mathbf{S}) \rightarrow \mathbf{Env}$$

La déclaration de constante présente trois cas syntaxiques : la liaison d'un identificateur individuel à sa valeur, la séquence de liaisons ointroduite par '`;`' et la liste de déclarations récursives introduite par `rec`. Dans le cas de la liaison individuelle, il suffit d'évaluer l'expression dans le contexte courant et à produire un environnement contenant uniquement la liaison de l'identificateur à la valeur ainsi obtenue. Pour le cas de la liste de liaisons, on évalue les deux parties de la liste pour obtenir deux environnements ρ_1 et ρ_2 qui sont ensuite combiner avec la fonction *combine* puisqu'on ne permet pas de lier deux fois le même identificateur dans une liste de liaisons.

Le cas des liaisons récursives est un peu plus complexe. Dans les liaisons récursives, on permet à des expressions de la liste de liaison de référencer des identificateurs qui doivent être liés par cette même liste de liaison. L'évaluation de l'expression devrait donc se faire dans l'environnement résultant du traitement de la liste de liaison. Une restriction est mise cependant : comme en Scheme, l'évaluation des expression ne doit pas exiger de déréférencere ces identificateurs mais seulement de les mentionner. Nous sommes clairement dans un cas de

calcul par point fixe. Il faut trouver une environnement ρ_1 tel que :

$$\rho_1 = \mathcal{DC}[\![dc]\!](overlay(\rho_1, \rho), \sigma)$$

où ρ et σ sont l'environnement et la mémoire externes dans lesquels l'expression `soit` est évaluée. La solution de cette équation est obtenue par calcul de point fixe, comme cela apparaît à la figure 6.5. En commençant avec l'environnement vide *void*, on va pouvoir évaluer les expressions et produire un nouvel environnement, disons ρ_1. Dans ρ_1, les identificateurs vont être liés à des expressions qui sont évaluée dans *void*. Toute référence à un identificateur résulte alors en une erreur. Si on répète à nouveau le traitement avec ρ_1, on va obtenir un troisième environnement ρ_2 dans lequel les expressions sont évaluées dans l'environnement ρ_1, c'est-à-dire qu'une référence à un identificateur ν va demander l'évaluation de l'expression e qui sera elle évaluée dans l'environnement vide. On va donc pouvoir se permettre un niveau de référence aux identificateurs. En continuant le processus, on va produire un environnement ρ_3 dans lequel on va pouvoir se permettre deux niveux de référence, puis quatre, et ainsi de suite. Cette séquence converge vers un environnement ρ qui est le point fixe de la déclaration récursive où on pourra se permettre n niveaux de référence aux identificateurs pour tout n fini. La fonction dont on prend le point fixe est de type **env** $\rightarrow$ **env**, son point fixe retourne donc une valeur du domaine **Env**.

Revenons maintenant à l'expression `soit`. Elle est traitée en calculant d'abord l'environnement résultant de la liste de liaisons, puis en évaluant l'expression de son corps dans le contexte de cet environnement et de la mémoire courante :

$$\begin{array}{l}\mathcal{E}[\![\texttt{(soit } dc \texttt{ dans } e\texttt{)}]\!] = \\ \quad \lambda(\rho, \sigma).\ \textbf{let } \rho_1 = \mathcal{DC}[\![dc]\!](\rho, \sigma) \\ \qquad\qquad \textbf{in } \mathcal{E}[\![e]\!](overlay(\rho_1, \rho), \sigma)\end{array}$$

Les déclarations de fonctions et de procédures sont toutes les deux introduites par des expressions. Le résultat de ces expressions doit être une fonction (au sens de domaine sémantique de fonctions) qui va réaliser le calcul exprimé par la fonction ou la procédure syntaxiques. Dans les deux cas, fonction et procédure, il faut assurer le passage des paramètres. Pour le réaliser, la fonction (sémantique) qui représente une fonction (syntaxique) ou une procédure doit prendre en paramètre une valeur du domaine **PV** et lier d'une manière ou d'une auter cette valeur au paramètre formel déclaré. Plusieurs cas peuvent se produire selon que le paramètre est passé par valeur ou par référence, et selon le type du paramètre. S'il est passé par valeur, il faut allouer un espace mémoire pour contenir cette valeur et lier le paramètre formel à l'adresse de cet espace mémoire dans l'environnement local. S'il est passé par référence, la valeur reçue sera une adresse et cette adresse sera simplement lié au paramètre formel dans l'environnement local. L'environnement local est ensuite combiner par la fonction *overlay* avec l'environnement de définition de la fonction ou de la procédure. La création de l'environnement local et la modification de l'état mémoire pour le passage de paramètre est réalisé par la fonction sémantique $\mathcal{DP}$ apparaissant à la figure 6.6.

Le passage par référence nécessite une allocation de mémoire. Cette allocation dépend du type du paramètre. Dans notre langage, un paramètre de type `bool` ou `num` nécessitera un seul mot mémoire, mais si nous voulions étendre le langage aux tableaux, on devrait allouer plusieurs mots mémoire pour un paramètre. L'allocation est donc réalisée par la fonction sémantique $\mathcal{T}$ définie à la figure 6.7. Elle prend une déclaration de type et une mémoire courante, et retourne une adresse du bloc de mémoire alloué et une mémoire dans laquelle l'espace alloué a été réservé (mais non initialisé).

$$\mathcal{DP} :: \text{DeclarationParametre} \to \mathbf{PV} \to \mathbf{S} \to (\mathbf{Env} \otimes \mathbf{S})$$

$$\mathcal{DP}[\![\texttt{val}\ \ \nu\ \ type]\!] =$$
$$\quad \lambda\pi.\lambda\sigma.\ \ \mathbf{let}\ (\iota, \sigma_1) = \mathcal{T}[\![type]\!](\sigma)$$
$$\qquad\qquad \mathbf{in}\ (binding(\nu, in\mathbf{DV}(\iota)), store(\iota\ |_{\mathbf{Loc}}, \pi, \sigma))$$

$$\mathcal{DP}[\![\texttt{var}\ \ \nu\ \ type]\!] = \ \lambda\pi.\lambda\sigma.(binding(\nu, \pi), \sigma)$$

$$\mathcal{LP} :: \text{DeclarationParametre} \to \mathbf{Env} \to \mathbf{S} \to \mathbf{S}$$

$$\mathcal{LP}[\![\texttt{val}\ \ \nu\ \ type]\!] =$$
$$\quad \lambda\rho.\lambda\sigma.\ \ \mathbf{let}\ \iota = bound(\nu, \rho)\ |_{\mathbf{LV}}$$
$$\qquad\qquad \mathbf{in}\ \mathcal{LT}[\![type]\!](\iota, \sigma)$$

$$\mathcal{LP}[\![\texttt{var}\ \ \nu\ \ type]\!] = \ \lambda\rho.\lambda\sigma.\sigma$$

FIG. 6.6 – Fonctions de valuation des déclarations de paramètres

$$\mathcal{T} :: \text{Type} \to \mathbf{S} \to (\mathbf{LV} \otimes \mathbf{S})$$

$$\mathcal{T}[\![\ \texttt{bool}\ ]\!] =$$
$$\quad \lambda\sigma.\ \ \mathbf{let}\ (l, \sigma_1) = allocation(\sigma)$$
$$\qquad\ \mathbf{in}\ (in\mathbf{LV}(l), \sigma_1)$$

$$\mathcal{T}[\![\ \texttt{num}\ ]\!] =$$
$$\quad \lambda\sigma.\ \ \mathbf{let}\ (l, \sigma_1) = allocation(\sigma)$$
$$\qquad\ \mathbf{in}\ (in\mathbf{LV}(l), \sigma_1)$$

$$\mathcal{LT} :: \text{Type} \to (\mathbf{LV} \otimes \mathbf{S}) \to \mathbf{S}$$

$$\mathcal{LT}[\![\ \texttt{bool}\ ]\!] =$$
$$\quad \lambda(\iota, \sigma).free(\iota\ |_{\mathbf{Loc}}, \sigma)$$

$$\mathcal{LT}[\![\ \texttt{num}\ ]\!] =$$
$$\quad \lambda(\iota, \sigma).free(\iota\ |_{\mathbf{Loc}}, \sigma)$$

FIG. 6.7 – Fonctions de valuation des déclarations de types

À la fin de l'exécution de la fonction ou de la procédure, il est bon d'indiquer que l'espace alloué doit être libéré. Cette libération va entraîner une modification de la mémoire, et elle dépend du type du paramètre. On confie donc ce travail aux fonctions sémantiques $\mathcal{LP}$ et $\mathcal{LT}$ qui tiennent compte du mode de passage du paramètre et du type déclaré.

Si on revient maintenant à la sémantique de la déclaration d'une fonction, il faut produire une fonction qui prend l'état mémoire lors de l'application de la fonction ainsi que la valeur du paramètre réel, et qui exécute la fonction. L'exécution de la fonction consiste d'abord à effectuer le passage du paramètre, puis d'exécuter l'expression qui forme le corps de la fonction et enfin de libérer l'espace éventuellement alloué pour le paramètre. Le résultat sera celui de l'évaluation du corps de la fonction :

$$\mathcal{E}[\![\texttt{(fun (}dp\texttt{) }e\texttt{)}]\!] =$$
$$\quad \lambda(\rho, \sigma).in\mathbf{EV}(\lambda\sigma_1.\lambda\pi.\ \ \mathbf{let*}\ (\rho_1, \sigma_2) = \mathcal{DP}[\![dp]\!](\pi)(\sigma_1)$$
$$\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{and}\ \varepsilon = \mathcal{R}[\![e]\!](overlay(\rho_1, \rho), \sigma_2)$$

$$\begin{array}{l}\textbf{and } \sigma_3 = \mathcal{LP}[\![dp]\!](\rho_1)(\sigma_2)\\ \textbf{in } \varepsilon\,|_{\mathbf{V}}\ )\end{array}$$

Pour une procédure, l'idée est similaire, à ceci près qu'il faut aussi passer l'état de l'entrée et de la sortie standard. L'exécution du corps de la procédure, qui est un énoncé, peut modifier l'état mémoire ou l'état de l'entrée et de la sortie standard ; le résultat de cette exécution sera donc un triplet état mémoire, entrée standard et sortie standard :

$$\begin{array}{l}\mathcal{E}[\![\texttt{(proc (} dp \texttt{) } c\texttt{)}]\!] =\\ \quad \lambda(\rho,\sigma).\mathit{in}\mathbf{EV}(\lambda\pi.\lambda(\texttt{in},\texttt{out},\sigma_1).\ \textbf{let*}\ (\rho_1,\sigma_2) = \mathcal{DP}[\![dp]\!](\pi)(\sigma_1)\\ \qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{and}\ (\texttt{in}_1,\texttt{out}_1,\sigma_3) = \mathcal{C}[\![c]\!](\mathit{overlay}(\rho_1,\rho),\texttt{in},\texttt{out},\sigma_2)\\ \qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{and}\ \sigma_4 = \mathcal{LP}[\![dp]\!](\rho_1)(\sigma_3)\\ \qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{in}\ (\texttt{in}_1,\texttt{out}_1,\sigma_4)\ )\end{array}$$

L'application de fonction est une expression, dans la mesure ou elle retourne un résultat. Par contre, l'application de procédure est un énoncé sur lequel nous revenons plus loin. L'application de fonction consiste à évaluer la fonction et son paramètre, puis à appliquer le résultat de la première au résultat de la seconde :

$$\begin{array}{l}\mathcal{E}[\![e\ \texttt{(}\ e_1\ \texttt{)}]\!] =\\ \quad \lambda(\rho,\sigma).\ \textbf{let}\ \varepsilon = \mathcal{R}[\![e]\!](\rho,\sigma)\\ \qquad\qquad\quad \textbf{and}\ \varepsilon_1 = \mathcal{E}[\![e_1]\!](\rho,\sigma)\\ \qquad\qquad\quad \textbf{in}\ \varepsilon\,|_{\mathbf{F}}\ (\sigma)(\varepsilon_1)\end{array}$$

Les équations sémantiques définissant les fonctions sémantiques $\mathcal{E}$ et $\mathcal{R}$ sont regroupées à la figure 6.8.

6.5 Programme et énoncés

Un programme est constitué d'une liste de déclarations de constantes, permettant de définir surtout les fonctions et les constantes globales du programme, et d'un énoncé ; évaluer un programme consiste donc à créer l'environnement contenant les liaisons des constantes, puis à évaluer l'énoncé dans le contexte de cet environnement. C'est aussi pour l'évaluation d'un programme que sont passés le fichier d'entrée et le fichier de sortie standards. La fonction sémantique $\mathcal{P}$ prend donc un programme et retourne une fonction qui prend en entrée un état initial du fichier d'entrée et du fichier de sortie, et retourne comme résultat l'état de ces deux fichiers à la fin du programme :

$$\begin{array}{l}\mathcal{P}[\![\texttt{prog}\ \ dc\ \ c]\!] =\\ \quad \lambda(\texttt{in},\texttt{out}).\ \textbf{let*}\ \rho = \mathcal{DC}[\![dc]\!](\mathit{void},\mathit{empty\text{-}store})\\ \qquad\qquad\quad \textbf{and}\ (\texttt{in}_1,\texttt{out}_1,\sigma) = \mathcal{C}[\![c]\!](\rho,\texttt{in},\texttt{out},\mathit{empty\text{-}store})\\ \qquad\qquad\quad \textbf{in}\ (\texttt{in}_1,\texttt{out}_1)\end{array}$$

L'énoncé du programme est évalué à partir d'une mémoire vide, comme il se doit, puisque les déclarations de constantes n'introduisent pas de variables. Si des variables sont nécessaires pour le programme, elles sont introduites dans l'énoncé en utilisant la forme syntaxique `begin` définie ci-bas.

Considérons maintenant les énonces. L'évaluation d'un énoncé se fait dans le contexte d'un environnement, d'un état mémoire, et de l'état des fichiers d'netrée et de sortie standards.

$\mathcal{L}$ :: Constantes $\rightarrow$ **V**

$\mathcal{L}[\![k]\!] = \ \textit{non-spécifié}$

$\mathcal{E}$ :: Expressions $\rightarrow$ (**Env** $\otimes$ **S**) $\rightarrow$ **EV**

$\mathcal{E}[\![k]\!] = \ \lambda(\rho,\sigma).\textit{in}\mathbf{EV}(\mathcal{L}[\![k]\!])$

$\mathcal{E}[\![\nu]\!] = \ \lambda(\rho,\sigma).\textit{bound}(\nu,\rho)$

$\mathcal{E}[\![om\ \ e]\!] = \ \lambda(\rho,\sigma).\ \mathbf{let}\ \textit{fnm} = \mathcal{OM}[\![om]\!]\ \mathbf{and}\ \varepsilon = \mathcal{R}[\![e]\!](\rho,\sigma)\ \mathbf{in}\ \textit{fnm}(\varepsilon)$

$\mathcal{E}[\![od\ \ e\ \ e_1]\!] =$
$\lambda(\rho,\sigma).\ \mathbf{let}\ \textit{fnd} = \mathcal{OD}[\![od]\!]$
$\mathbf{and}\ \varepsilon = \mathcal{R}[\![e]\!](\rho,\sigma)$
$\mathbf{and}\ \varepsilon_1 = \mathcal{R}[\![e_1]\!](\rho,\sigma)$
$\mathbf{in}\ \textit{fnd}(\varepsilon,\varepsilon_1)$

$\mathcal{E}[\![\texttt{(choix}\ e\ \ e_1\ \ e_2\texttt{)}]\!] =$
$\lambda(\rho,\sigma).\ \mathbf{let}\ \varepsilon = \mathcal{R}[\![e]\!](\rho,\sigma)$
$\mathbf{in}\ \ \mathbf{if}\ \varepsilon\ |_{\mathbf{V}}|_{\mathbf{T}}$
$\mathbf{then}\ \mathcal{E}[\![e_1]\!](\rho,\sigma)$
$\mathbf{else}\ \mathcal{E}[\![e_2]\!](\rho,\sigma)$
$\mathbf{endif}$

$\mathcal{E}[\![\texttt{(soit}\ dc\ \texttt{dans}\ e\texttt{)}]\!] =$
$\lambda(\rho,\sigma).\ \mathbf{let}\ \rho_1 = \mathcal{DC}[\![dc]\!](\rho,\sigma)$
$\mathbf{in}\ \mathcal{E}[\![e]\!](\textit{overlay}(\rho_1,\rho),\sigma)$

$\mathcal{E}[\![\texttt{(fun (}\ dp\ \texttt{)}\ e\texttt{)}]\!] =$
$\lambda(\rho,\sigma).\textit{in}\mathbf{EV}(\lambda\sigma_1.\lambda\pi.\ \ \mathbf{let*}\ (\rho_1,\sigma_2) = \mathcal{DP}[\![dp]\!](\pi)(\sigma_1)$
$\mathbf{and}\ \varepsilon = \mathcal{R}[\![e]\!](\textit{overlay}(\rho_1,\rho),\sigma_2)$
$\mathbf{and}\ \sigma_3 = \mathcal{LP}[\![dp]\!](\rho_1)(\sigma_2)$
$\mathbf{in}\ \varepsilon\ |_{\mathbf{V}}\ )$

$\mathcal{E}[\![\texttt{(proc (}\ dp\ \texttt{)}\ c\texttt{)}]\!] =$
$\lambda(\rho,\sigma).\textit{in}\mathbf{EV}(\lambda\pi.\lambda(\texttt{in},\texttt{out},\sigma_1).\ \ \mathbf{let*}\ (\rho_1,\sigma_2) = \mathcal{DP}[\![dp]\!](\pi)(\sigma_1)$
$\mathbf{and}\ (\texttt{in}_1,\texttt{out}_1,\sigma_3) = \mathcal{C}[\![c]\!](\textit{overlay}(\rho_1,\rho),\texttt{in},\texttt{out},\sigma_2)$
$\mathbf{and}\ \sigma_4 = \mathcal{LP}[\![dp]\!](\rho_1)(\sigma_3)$
$\mathbf{in}\ (\texttt{in}_1,\texttt{out}_1,\sigma_4)\ )$

$\mathcal{E}[\![e\ \texttt{(}\ e_1\ \texttt{)}]\!] =$
$\lambda(\rho,\sigma).\ \mathbf{let}\ \varepsilon = \mathcal{R}[\![e]\!](\rho,\sigma)$
$\mathbf{and}\ \varepsilon_1 = \mathcal{E}[\![e_1]\!](\rho,\sigma)$
$\mathbf{in}\ \varepsilon\ |_{\mathbf{F}}\ (\sigma)(\varepsilon_1)$

$\mathcal{R}$ :: Expressions $\rightarrow$ (**Env** $\otimes$ **S**) $\rightarrow$ **EV**

$\mathcal{R}[\![e]\!] = \ \lambda(\rho,\sigma).\ \mathbf{let}\ \varepsilon = \mathcal{E}[\![e]\!](\rho,\sigma)\ \mathbf{in}\ \mathbf{if}\ \textit{is}\mathbf{LV}\ ?(\varepsilon)\ \mathbf{then}\ \textit{stored}(\varepsilon\ |_{\mathbf{LV}}|_{\mathbf{Loc}},\sigma)\ \mathbf{else}\ \varepsilon\ \mathbf{endif}$

FIG. 6.8 – Fonctions de valuation des littéraux et des expressions

Le résultat retourné doit contenir l'état mémoire après l'énoncé, ainsi que l'état des fichiers d'entrée et de sortie standards. La fonction sémantique $\mathcal{C}$ dont le type est :

$$\mathcal{C} :: \text{Commandes} \rightarrow (\mathbf{Env} \otimes \mathbf{In} \otimes \mathbf{Out} \otimes \mathbf{S}) \rightarrow (\mathbf{In} \otimes \mathbf{Out} \otimes \mathbf{S})$$

L'énoncé le plus simple est `skip` qui ne change rien dans l'état du programme :

$$\mathcal{C}[\![\texttt{skip}]\!] = \lambda(\rho, \texttt{in}, \texttt{out}, \sigma).(\texttt{in}, \texttt{out}, \sigma)$$

Pour l'énoncé d'affectation, il faut évaluer l'expression en partie gauche de l'affectation, qui doit nécessairement retouner une valeur à gauche, puis l'expression en partie droite. La valeur de la seconde sera ensuite liée à l'adresse obteur par la première dans la mémoire courante :

$$\begin{array}{l}\mathcal{C}[\![e \texttt{ := } e_1]\!] = \\ \quad \lambda(\rho, \texttt{in}, \texttt{out}, \sigma).\ \textbf{let}\ \varepsilon = \mathcal{E}[\![e]\!](\rho, \sigma) \\ \qquad\qquad\qquad\ \textbf{and}\ \varepsilon_1 = \mathcal{R}[\![e_1]\!](\rho, \sigma) \\ \qquad\qquad\qquad\ \textbf{in}\ (\texttt{in}, \texttt{out}, \mathit{store}(\varepsilon\ |_{\mathbf{LV}}|_{\mathbf{Loc}}, \varepsilon_1, \sigma))\end{array}$$

Pour l'énoncé de composition séquentielle, il suffit d'évaluer la première partie de la séquence pour obtenir un nouvel état mémoire et état des fichiers d'entrée et sortie, qui seront utilisés pour évaluer la seconde partie de la séquence :

$$\begin{array}{l}\mathcal{C}[\![c \texttt{ ; } c_1]\!] = \\ \quad \lambda(\rho, \texttt{in}, \texttt{out}, \sigma).\ \textbf{let}\ (\texttt{in}_1, \texttt{out}_1, \sigma_1) = \mathcal{C}[\![c]\!](\rho, \texttt{in}, \texttt{out}, \sigma) \\ \qquad\qquad\qquad\ \textbf{in}\ \mathcal{C}[\![c_1]\!](\rho, \texttt{in}_1, \texttt{out}_1, \sigma_1)\end{array}$$

L'évaluation de l'énoncé d'alternative est très similaire à celui de l'expression `choix` à ceci près que les parties alors et sinon sont des énoncés :

$$\begin{array}{l}\mathcal{C}[\![\texttt{if } e \texttt{ } c \texttt{ } c_1]\!] = \\ \quad \lambda(\rho, \texttt{in}, \texttt{out}, \sigma).\ \textbf{let}\ \varepsilon = \mathcal{E}[\![e]\!](\rho, \sigma) \\ \qquad\qquad\qquad\ \textbf{in}\ \ \textbf{if}\ \varepsilon\ |_{\mathbf{V}}|_{\mathbf{T}} \\ \qquad\qquad\qquad\qquad \textbf{then}\ \mathcal{C}[\![c]\!](\rho, \texttt{in}, \texttt{out}, \sigma) \\ \qquad\qquad\qquad\qquad \textbf{else}\ \mathcal{C}[\![c_1]\!](\rho, \texttt{in}, \texttt{out}, \sigma) \\ \qquad\qquad\qquad\qquad \textbf{endif}\end{array}$$

L'énoncé de répétition `tantque`, comme nous l'avons vu au chapitre 4, nécessite de trouver le fonction f telle que :

$$\begin{array}{rl}f((\texttt{in}, \texttt{out}, \sigma)) = & \textbf{let}\ \varepsilon = \mathcal{E}[\![e]\!](\rho, \sigma) \\ & \textbf{in}\ \ \textbf{if}\ \varepsilon\ |_{\mathbf{V}}|_{\mathbf{T}} \\ & \qquad \textbf{then}\ f(\mathcal{C}[\![c]\!](\rho, \texttt{in}, \texttt{out}, \sigma)) \\ & \qquad \textbf{else}\ (\texttt{in}, \texttt{out}, \sigma) \\ & \qquad \textbf{endif}\ \ )\end{array}$$

où ρ est l'environnement dans lequel est évaluée la répétition. Il s'agit donc de trouver la fonction f qui, étant donné un état mémoire initial σ et les états initiaux de l'entrée et de la sortie standards `in` et `out`, va évaluer la condition de la répétition dans l'état (ρ, σ), et si la condition est vraie va évaluer le corps de la répétition et se rappeler récursivement sur l'état résultat de cette évaluation. Pour obtenir cette fonction, on prend le point fixe de la fonction prenant un triplet $(\texttt{in}, \texttt{out}, \sigma)$ et retourne un autre triplet résultant de l'exécution de la répétition.[4] On obtient alors la définition suivante pour la sémantique de la répétition :

$$\begin{array}{l}\mathcal{C}[\![\texttt{tantque } e \texttt{ faire } c]\!] = \\ \quad \lambda(\rho, \texttt{in}, \texttt{out}, \sigma).\mathbf{fix}(\lambda f.\lambda tmp.\ \ \textbf{let*}\ (\texttt{in}_1, \texttt{out}_1, \sigma_1) = tmp\end{array}$$

[4]Notre opérateur de point fixe **fix** prend une fonction $f : \mathbf{D} \to \mathbf{D}$ qui prend un seul paramètre et retourne un seul résultat. On utilise donc un triplet plutôt que trois paramètres et trois résultats (ce qui de toutes façons poseraient problème).

$$\mathcal{DV} :: \text{DeclarationVariables} \to \mathbf{S} \to (\mathbf{Env} \otimes \mathbf{S})$$

$$\mathcal{DV}[\![\texttt{var}\ \ \nu\ \mathit{type}]\!] =$$
$$\quad \lambda\sigma.\ \mathbf{let}\ (\iota, \sigma_1) = \mathcal{T}[\![\mathit{type}]\!](\sigma)$$
$$\qquad \mathbf{in}\ (\mathit{binding}(\nu, \mathit{in}\mathbf{DV}(\iota)), \sigma_1)$$

$$\mathcal{DV}[\![\mathit{dv}\ ,\ \mathit{dv}_1]\!] =$$
$$\quad \lambda\sigma.\ \mathbf{let*}\ (\rho, \sigma_1) = \mathcal{DV}[\![\mathit{dv}]\!](\sigma)$$
$$\qquad \mathbf{and}\ (\rho_1, \sigma_2) = \mathcal{DV}[\![\mathit{dv}_1]\!](\sigma_1)$$
$$\qquad \mathbf{in}\ (\mathit{combine}(\rho, \rho_1), \sigma_2)$$

$$\mathcal{LV} :: \text{DeclarationVariables} \to (\mathbf{Env} \otimes \mathbf{S}) \to \mathbf{S}$$

$$\mathcal{LV}[\![\texttt{var}\ \ \nu\ \mathit{type}]\!] =$$
$$\quad \lambda(\rho, \sigma).\ \mathbf{let}\ \iota = \mathit{bound}(\nu, \rho)\ |_{\mathbf{LV}}$$
$$\qquad \mathbf{in}\ \mathcal{LT}[\![\mathit{type}]\!](\iota, \sigma)$$

$$\mathcal{LV}[\![\mathit{dv}\ ,\ \mathit{dv}_1]\!] = \lambda(\rho, \sigma).\ \mathbf{let}\ \sigma_1 = \mathcal{LV}[\![\mathit{dv}]\!](\rho, \sigma)\ \mathbf{in}\ \mathcal{LV}[\![\mathit{dv}_1]\!](\rho, \sigma_1)$$

FIG. 6.9 – Fonctions de valuation des déclarations de variables

$$\begin{array}{l}
\quad \mathbf{and}\ \varepsilon = \mathcal{E}[\![e]\!](\rho, \sigma_1) \\
\mathbf{in}\ \ \mathbf{if}\ \varepsilon\ |_{\mathbf{V}}|_{\mathbf{T}} \\
\qquad \mathbf{then}\ f(\mathcal{C}[\![c]\!](\rho, \mathtt{in}_1, \mathtt{out}_1, \sigma_1)) \\
\qquad \mathbf{else}\ \mathit{tmp} \\
\qquad \mathbf{endif}\ \ )((\mathtt{in}, \mathtt{out}, \sigma))
\end{array}$$

L'énoncé dit bloc déclare une liste de variables et évalue un énoncé dans le contexte de ces déclarations. Il faut donc d'abord traiter les déclarations de variables pour allouer l'espace nécessaire. Comme dans le cas des déclarations de paramètres, cette partie est déléguée à la fonction sémantique $\mathcal{DV}$ dont le rôle est d'allouer la mémoire en fonction du type des varaibles et de retourner un environnement liant identificateurs aux adresses et une mémoire modifiée où l'espace alloué est réservé mais non initialisé. La définition de $\mathcal{DV}$ apparaît à la figure 6.9. À la fin de l'exécution du corps, comme à la fin de l'exécution d'une procédure ou d'un méthode, l'espace alloué doit être libéré, ce qui est réalisé par la fonction sémantique $\mathcal{LV}$, elle aussi dépendant du type et définie à la figure 6.9. La sémantique de l'énoncé `begin` est donc :

$$\begin{array}{l}
\mathcal{C}[\![\texttt{begin}\ \mathit{dv}\ \texttt{;}\ c\ \texttt{end}]\!] = \\
\quad \lambda(\rho, \mathtt{in}, \mathtt{out}, \sigma).\ \mathbf{let*}\ (\rho_1, \sigma_1) = \mathcal{DV}[\![\mathit{dv}]\!](\sigma) \\
\qquad\qquad\qquad\quad \mathbf{and}\ (\mathtt{in}_1, \mathtt{out}_1, \sigma_2) = \mathcal{C}[\![c]\!](\rho_1, \mathtt{in}, \mathtt{out}, \sigma_1) \\
\qquad\qquad\qquad\quad \mathbf{and}\ \sigma_3 = \mathcal{LV}[\![\mathit{dv}]\!](\rho_1, \sigma_2) \\
\qquad\qquad\qquad\quad \mathbf{in}\ (\mathtt{in}_1, \mathtt{out}_1, \sigma_3)
\end{array}$$

L'application de procédure est similaire à l'application de fonction, à ceci près que le corps de la procédure est un énoncé et doit donc être évalué dans un contexte incluant l'état mémoire et les entréée et sortie standards, et retourne un nouvel état :

$$\begin{array}{l}
\mathcal{C}[\![e\ \texttt{(}\ e_1\texttt{)}]\!] = \\
\quad \lambda(\rho, \mathtt{in}, \mathtt{out}, \sigma).\ \mathbf{let}\ \varepsilon = \mathcal{R}[\![e]\!](\rho, \sigma) \\
\qquad\qquad\qquad\quad \mathbf{and}\ \varepsilon_1 = \mathcal{E}[\![e_1]\!](\rho, \sigma) \\
\qquad\qquad\qquad\quad \mathbf{in}\ \varepsilon\ |_{\mathbf{P}}\ (\varepsilon_1)(\mathtt{in}, \mathtt{out}, \sigma)
\end{array}$$

L'énoncé de lecture évalue l'expression argument qui doit retourner une valeur à gauche. Cette valeur à gauche est utilisée pour aller ranger en mémoire la prochaine valeur apparaissant dans l'entrée standard, qui en est extraite pour produire un nouvel état de l'entrée standard :

$$\begin{array}{l}\mathcal{C}[\![\texttt{lire}\ \ e]\!] = \\ \quad \lambda(\rho, \texttt{in}, \texttt{out}, \sigma).\ \ \textbf{let}\ \varepsilon = \mathcal{E}[\![e]\!](\rho, \sigma) \\ \qquad\qquad\qquad\quad \textbf{in}\ (\texttt{in} \dagger 1, \texttt{out}, \mathit{store}(\varepsilon\ |_{\mathbf{LV}}|_{\mathbf{Loc}}, \texttt{in}\downarrow_1, \sigma))\end{array}$$

L'énoncé d'écriture évalue l'expression et place son résultat à la fin de la sortie standard, ce qui produit un nouvel état pour cette sortie standard :

$$\begin{array}{l}\mathcal{C}[\![\texttt{ecrire}\ \ e]\!] = \\ \quad \lambda(\rho, \texttt{in}, \texttt{out}, \sigma).\ \ \textbf{let}\ \varepsilon = \mathcal{R}[\![e]\!](\rho, \sigma) \\ \qquad\qquad\qquad\quad \textbf{in}\ (\texttt{in}, \texttt{out}\S\langle\varepsilon\rangle, \sigma)\end{array}$$

Les équations sémantiques définissant les fonctions sémantiques $\mathcal{P}$ et $\mathcal{C}$ sont regroupées à la figure 6.10.

$\mathcal{P}$ :: Programme $\to$ (**In** $\otimes$ **Out**) $\to$ (**In** $\otimes$ **Out**)

$$\mathcal{P}[\![\texttt{prog}\ \ dc\ \ c]\!] =$$
$$\lambda(\texttt{in},\texttt{out}).\ \ \textbf{let*}\ \rho = \mathcal{DC}[\![dc]\!](\mathit{void}, \mathit{empty\text{-}store})$$
$$\textbf{and}\ (\texttt{in}_1, \texttt{out}_1, \sigma) = \mathcal{C}[\![c]\!](\rho, \texttt{in}, \texttt{out}, \mathit{empty\text{-}store})$$
$$\textbf{in}\ (\texttt{in}_1, \texttt{out}_1)$$

$\mathcal{C}$ :: Commandes $\to$ (**Env** $\otimes$ **In** $\otimes$ **Out** $\otimes$ **S**) $\to$ (**In** $\otimes$ **Out** $\otimes$ **S**)

$$\mathcal{C}[\![\texttt{skip}]\!] = \ \lambda(\rho, \texttt{in}, \texttt{out}, \sigma).(\texttt{in}, \texttt{out}, \sigma)$$

$$\mathcal{C}[\![e\ \texttt{:=}\ e_1]\!] =$$
$$\lambda(\rho, \texttt{in}, \texttt{out}, \sigma).\ \ \textbf{let}\ \varepsilon = \mathcal{E}[\![e]\!](\rho, \sigma)$$
$$\textbf{and}\ \varepsilon_1 = \mathcal{R}[\![e_1]\!](\rho, \sigma)$$
$$\textbf{in}\ (\texttt{in}, \texttt{out}, \mathit{store}(\varepsilon\ |_{\mathbf{LV}}|_{\mathbf{Loc}}, \varepsilon_1, \sigma))$$

$$\mathcal{C}[\![c\ \texttt{;}\ c_1]\!] =$$
$$\lambda(\rho, \texttt{in}, \texttt{out}, \sigma).\ \ \textbf{let}\ (\texttt{in}_1, \texttt{out}_1, \sigma_1) = \mathcal{C}[\![c]\!](\rho, \texttt{in}, \texttt{out}, \sigma)$$
$$\textbf{in}\ \mathcal{C}[\![c_1]\!](\rho, \texttt{in}_1, \texttt{out}_1, \sigma_1)$$

$$\mathcal{C}[\![\texttt{if}\ \ e\ \ c\ \ c_1]\!] =$$
$$\lambda(\rho, \texttt{in}, \texttt{out}, \sigma).\ \ \textbf{let}\ \varepsilon = \mathcal{E}[\![e]\!](\rho, \sigma)$$
$$\textbf{in}\ \ \textbf{if}\ \varepsilon\ |_{\mathbf{V}}|_{\mathbf{T}}$$
$$\textbf{then}\ \mathcal{C}[\![c]\!](\rho, \texttt{in}, \texttt{out}, \sigma)$$
$$\textbf{else}\ \mathcal{C}[\![c_1]\!](\rho, \texttt{in}, \texttt{out}, \sigma)$$
$$\textbf{endif}$$

$$\mathcal{C}[\![\texttt{tantque}\ e\ \ \texttt{faire}\ c]\!] =$$
$$\lambda(\rho, \texttt{in}, \texttt{out}, \sigma).\textbf{fix}(\lambda f.\lambda \mathit{tmp}.\ \ \textbf{let*}\ (\texttt{in}_1, \texttt{out}_1, \sigma_1) = \mathit{tmp}$$
$$\textbf{and}\ \varepsilon = \mathcal{E}[\![e]\!](\rho, \sigma_1)$$
$$\textbf{in}\ \ \textbf{if}\ \varepsilon\ |_{\mathbf{V}}|_{\mathbf{T}}$$
$$\textbf{then}\ f(\mathcal{C}[\![c]\!](\rho, \texttt{in}_1, \texttt{out}_1, \sigma_1))$$
$$\textbf{else}\ \mathit{tmp}$$
$$\textbf{endif}\ \ \ )((\texttt{in}, \texttt{out}, \sigma))$$

$$\mathcal{C}[\![\texttt{begin}\ dv\ \texttt{;}\ c\ \texttt{end}]\!] =$$
$$\lambda(\rho, \texttt{in}, \texttt{out}, \sigma).\ \ \textbf{let*}\ (\rho_1, \sigma_1) = \mathcal{DV}[\![dv]\!](\sigma)$$
$$\textbf{and}\ (\texttt{in}_1, \texttt{out}_1, \sigma_2) = \mathcal{C}[\![c]\!](\rho_1, \texttt{in}, \texttt{out}, \sigma_1)$$
$$\textbf{and}\ \sigma_3 = \mathcal{LV}[\![dv]\!](\rho_1, \sigma_2)$$
$$\textbf{in}\ (\texttt{in}_1, \texttt{out}_1, \sigma_3)$$

$$\mathcal{C}[\![e\ \texttt{(}\ e_1\texttt{)}]\!] =$$
$$\lambda(\rho, \texttt{in}, \texttt{out}, \sigma).\ \ \textbf{let}\ \varepsilon = \mathcal{R}[\![e]\!](\rho, \sigma)$$
$$\textbf{and}\ \varepsilon_1 = \mathcal{E}[\![e_1]\!](\rho, \sigma)$$
$$\textbf{in}\ \varepsilon\ |_{\mathbf{P}}\ (\varepsilon_1)(\texttt{in}, \texttt{out}, \sigma)$$

$$\mathcal{C}[\![\texttt{lire}\ \ e]\!] =$$
$$\lambda(\rho, \texttt{in}, \texttt{out}, \sigma).\ \ \textbf{let}\ \varepsilon = \mathcal{E}[\![e]\!](\rho, \sigma)$$
$$\textbf{in}\ (\texttt{in} \dagger 1, \texttt{out}, \mathit{store}(\varepsilon\ |_{\mathbf{LV}}|_{\mathbf{Loc}}, \texttt{in}{\downarrow_1}, \sigma))$$

$$\mathcal{C}[\![\texttt{ecrire}\ \ e]\!] =$$
$$\lambda(\rho, \texttt{in}, \texttt{out}, \sigma).\ \ \textbf{let}\ \varepsilon = \mathcal{R}[\![e]\!](\rho, \sigma)$$
$$\textbf{in}\ (\texttt{in}, \texttt{out}\S\langle\varepsilon\rangle, \sigma)$$

FIG. 6.10 – Fonctions de valuation des énoncés et du programme

Chapitre 7

Applications à la définition de la sémantique des langages à objets

Le présent chapitre introduit un langage minimal représentatif de l'essentiel du modèle de programmation par prototypes. Ce langage est nommé LIEBERMAN, car basé en grande partie sur la proposition originale de ce dernier [Lie86] bien qu'intégrant également plusieurs aspects du langage SELF [USC+91]. Dans un premier temps, nous définissons la syntaxe abstraite du langage. Puis, nous introduisons et commentons une sémantique dénotationnelle pour ce langage.

7.1 Définition du langage LIEBERMAN

Le langage LIEBERMAN est basé sur les prototypes, définis comme des collections de champs nommés. Chaque champ contient une valeur qui peut être une constante de base (entiers, réels, caractères, etc.) ou un identificateur d'objet, ce dernier pouvant pointer vers un objet méthode ou un objet standard selon la tradition de Smalltalk poursuivie par Self. La seule façon d'activer un prototype est de lui envoyer un message, le sélecteur duquel devant alors correspondre à l'un des noms des champs du receveur. LIEBERMAN offre une délégation implicite simple par laquelle un prototype identifie un parent auquel seront délégués les messages dont le sélecteur ne correspond à aucun des noms de ses champs. Le lien entre un prototype et son parent n'est pas accessible au programmeur. Le principal moyen pour créer un nouveau prototype est de cloner un prototype existant ; cependant, il existe une primitive permettant de créer un nouveau prototype comme fils d'un prototype existant en spécifiant l'ensemble de ses champs (noms et valeurs).

La syntaxe abstraite du langage apparaît à la figure 7.1. LIEBERMAN possède des expressions traditionnelles comme les constantes (valeurs de vérité, numéraux, caractères, chaînes de caractères et symboles), des identificateurs (pour accéder aux variables liées par l'expression `let`), des expressions de liaison `let`, des expressions de mutation `set` (pour changer la valeur des variables liées par l'expression `let`), et des expressions d'alternative `if`.

En ce qui concerne la programmation par objets, l'envoi de message est accompli par les expressions `send` (send $e\ q\ e^*$) qui envoient un message de sélecteur q à un objet dénoté par e avec les arguments dénotés par la liste d'expressions e^*. Il existe également des expressions `self` et `super` qui correspondent aux traditionnels envois de messages au receveur courant et au parent du prototype courant. La création d'un objet se fait principalement par l'expres-

sion (clone e), qui réalise une copie superficielle de l'objet dénoté par e et qui retourne le nouvel objet comme résultat. Une autre expression pour créer un objet est le `newInitials`, (newInitials e (q^*) (e^*)), qui crée un nouvel objet dont les noms de champs sont dénotés par la liste de symboles q^* et dont les valeurs de champs sont dénotées par la liste d'expressions e^* ; ce nouvel objet, qui est retourné comme résultat de l'expression, a pour parent l'objet dénoté par l'expression e. L'expression `root` retourne comme résultat l'objet premier du système, racine de l'arbre de délégation. Enfin, LIEBERMAN possède une expression `method`, similaire aux λ-expression des langages fonctionnels, pour créer les méthodes.

7.2 Dénotations à la Cook et Palsberg

Dans les sections suivantes, nous proposons une sémantique dénotationnelle pour ce langage minimal. Pour cela, nous utilisons la sémantique des enveloppeurs (*wrappers*) introduite par Cook et Palsberg [CP89], qui modélise les objets en utilisant un point fixe. Soit un objet `o1` représenté par une fonction des sélecteurs dans les valeurs (à définir plus tard), dont le domaine sémantique est défini comme étant **Behavior**. L'idée est d'obtenir `o1` en prenant le point fixe d'un *générateur d'objets*, une fonction de la forme (λself....), dont le domaine est **Behavior** $\rightarrow$ **Behavior**. Lorsqu'intervient l'héritage ou la délégation, l'objet fils est modélisé comme une fonction de la forme (λself.λsuper....), où l'argument *super* correspond à l'objet parent. Une telle fonction est appelée un *enveloppeur d'objet* (*object wrapper*), dont le domaine est **Behavior** $\rightarrow$ **Behavior** $\rightarrow$ **Behavior** (La figure 7.1 résume les domaines sémantiques pour les objets en LIEBERMAN).

L'application d'un enveloppeur crée un objet `o1` de parent `o2`. L'enveloppeur de `o1` est appliqué au générateur de `o2` en distribuant d'abord un nouveau `self` à l'enveloppeur et au générateur, puis en passant le générateur ainsi lié à l'enveloppeur comme *super*. Les deux composants résultant sont alors combinés de telle façon qu'un message non-répondu par l'enveloppeur maintenant lié de `o1` soit réexpedié vers le générateur maintenant lié de `o2`. L'application d'un enveloppeur est définie par l'opération $\triangleright$ prenant un enveloppeur et un générateur puis retournant un générateur, tandis que la combinaison d'objets biaisée vers l'objet de gauche est définie par l'opération $\odot$:

$b \odot b_1 =$
  $\lambda q.$ **let** $tmp = b(q)$
    in **if** is**O** $?(tmp)$
      then $b_1(q)$
      else tmp
      endif

$\Omega \triangleright \Gamma = \lambda\phi.$ **let** $b = \Gamma(\phi)$ **in** $\Omega(\phi)(b) \odot b$

Pour plus d'informations sur la sémantique des enveloppeurs pour les objets, le lecteur curieux se réfèrera à l'article de Cook et Palsberg [CP89] et à celui de Hense [Hen91].

7.3 Sémantique du langage LIEBERMAN

Dans ses grandes lignes, notre sémantique est écrite en style direct et elle est construite autour de quelques domaines sémantiques importants. Dans les pages qui suivent, nous supposons que le lecteur est familier avec les concepts de la programmation par objets et possède une

Domaines syntaxiques :

$$
\begin{array}{lcl}
k & \in & \textit{Constantes} \\
\nu & \in & \textit{Identificateurs} \\
e & \in & \textit{Expressions} \\
e & ::= & k \mid \nu \mid (\text{send } e\ q\ e^*) \mid (\text{self } q\ e^*) \mid (\text{super } q\ e^*) \mid (\text{method } (q^*)\ e^*) \mid \\
 & & (\text{clone } e) \mid (\text{newInitials } e\ (\nu^*)\ (e^*)) \mid (\text{let } (\nu\ e)\ e^*) \mid (\text{if } e\ e\ e) \mid \\
 & & (\text{set!}\ \nu\ e) \mid (\text{root}) \\
k & ::= & \texttt{\#t} \mid \texttt{\#f} \mid \textit{Num} \mid \textit{Char} \mid \textit{String} \mid \textit{Sym}
\end{array}
$$

Domaines sémantiques :

Domaines pour les littéraux et les domaines sémantiques caractéristiques :

$$
\begin{array}{rcll}
\tau & \in & \mathbf{T} & = \{\bot_{\mathbf{T}}, \mathbf{t}\text{rue}, \mathbf{f}\text{alse}\} \\
n & \in & \mathbf{N}\text{um} & = n\text{on-spécifié} \\
c & \in & \mathbf{C}\text{har} & = n\text{on-spécifié} \\
s & \in & \mathbf{S}\text{tring} & = n\text{on-spécifié} \\
q & \in & \mathbf{S}\text{ym} & = n\text{on-spécifié} \\
\upsilon & \in & \mathbf{V} & = \mathbf{T} \oplus \mathbf{N}\text{um} \oplus \mathbf{C}\text{har} \oplus \mathbf{S}\text{tring} \oplus \mathbf{S}\text{ym} \\
\vartheta & \in & \mathbf{R}\text{V} & = \mathbf{V} \oplus \mathbf{O}\text{op} \\
\iota & \in & \mathbf{L}\text{V} & = \mathbf{L}\text{oc} \\
\delta & \in & \mathbf{D}\text{V} & = \mathbf{E}\text{V} = \mathbf{R}\text{V} \oplus \mathbf{L}\text{V} \\
\pi & \in & \mathbf{P}\text{V} & = \mathbf{R}\text{V} \\
\omega & \in & \mathbf{S}\text{V} & = \mathbf{R}\text{V} \oplus \mathbf{O}\text{mV}
\end{array}
$$

Domaines sémantiques pour les environnements et les mémoires :

$$
\begin{array}{rcll}
t & \in & \mathbf{O} & = \{\bot_{\mathbf{O}}, \top\} \\
\nu & \in & \mathbf{I}\text{de} & = n\text{on-spécifié} \\
p & \in & \mathbf{P}\text{ointer} & = n\text{on-spécifié} \\
l & \in & \mathbf{L}\text{oc} & = \mathbf{P}\text{ointer} \oplus \mathbf{O} \\
\rho & \in & \mathbf{E}\text{nv} & = \mathbf{I}\text{de} \to (\mathbf{L}\text{oc} \oplus \mathbf{T}) \\
\sigma & \in & \mathbf{S} & = \mathbf{L}\text{oc} \to (\mathbf{S}\text{V} \oplus \mathbf{T})
\end{array}
$$

Domaines sémantiques pour les objets :

$$
\begin{array}{rcll}
\Delta & \in & \mathbf{D}\text{ict} & = \mathbf{S}\text{ym} \to (\mathbf{L}\text{oc} \oplus \mathbf{T}) \\
f & \in & \mathbf{F}\text{un} & = \mathbf{S} \to \mathbf{P}\text{V}^* \to (\mathbf{E}\text{V} \otimes \mathbf{S}) \\
\gamma & \in & \mathbf{M}\text{eth} & = (\mathbf{B}\text{ehavior} \otimes \mathbf{B}\text{ehavior}) \to \mathbf{F}\text{un} \\
b & \in & \mathbf{B}\text{ehavior} & = \mathbf{S}\text{ym} \to ((\mathbf{T} \to \mathbf{F}\text{un}) + \mathbf{O}) \\
\Gamma & \in & \mathbf{G}\text{enerator} & = \mathbf{B}\text{ehavior} \to \mathbf{B}\text{ehavior} \\
mega & \in & \mathbf{W}\text{rapper} & = \mathbf{B}\text{ehavior} \to \mathbf{B}\text{ehavior} \to \mathbf{B}\text{ehavior} \\
\varrho & \in & \mathbf{O}\text{op} & = \mathbf{P}\text{ointer} \oplus \mathbf{O} \\
o & \in & \mathbf{O}\text{bject} & = \mathbf{B}\text{ehavior} \times \mathbf{G}\text{enerator} \times \mathbf{S}\text{ym}^* \times \mathbf{L}\text{oc}^* \times \mathbf{G}\text{enerator} \\
\alpha & \in & \mathbf{O}\text{mV} & = \mathbf{O}\text{bject} \oplus \mathbf{M}\text{eth}
\end{array}
$$

FIG. 7.1 – Domaines syntaxiques et sémantiques pour le langage LIEBERMAN

connaissance minimale de la sémantique dénotationnelle. Parce que le langage LIEBERMAN va servir de base à notre comparaison de différentes formes de programmation centrée sur les objets concrets, nous introduisons maintenant l'ensemble de la sémantique ; par la suite, nous nous contenterons de présenter les différences par rapport à cette sémantique de base.

Les fonctions de valuation pour les expressions (voir la figure 7.2) prennent une expression, un argument *self* et un argument *super* consignant le contexte courant pour traiter les envois

$$\mathcal{L} :: \mathit{C}\text{onstantes} \to \mathbf{V}$$
$$\mathcal{L}[\![k]\!] = \mathit{unspecified}$$
$$\mathcal{E} :: \mathit{E}\text{xpressions} \to (\mathbf{B}\text{ehavior} \otimes \mathbf{B}\text{ehavior}) \to (\mathbf{E}\text{nv} \otimes \mathbf{S}) \to (\mathbf{E}\text{V} \otimes \mathbf{S})$$
$$\mathcal{E}[\![k]\!] = \lambda(\phi,\varphi).\lambda(\rho,\sigma).(\mathit{in}\mathbf{EV}(\mathit{in}\mathbf{RV}(\mathcal{L}[\![k]\!])),\sigma)$$
$$\mathcal{E}[\![\nu]\!] = \lambda(\phi,\varphi).\lambda(\rho,\sigma).(\mathit{in}\mathbf{EV}(\mathit{in}\mathbf{LV}(\rho(\nu)\mid_{\mathbf{Loc}})),\sigma)$$
$$\mathcal{E}[\![\texttt{(let (}\nu\ e\texttt{)}\ e^*\texttt{)}]\!] =$$
$$\lambda(\phi,\varphi).\lambda(\rho,\sigma).\ \mathbf{let*}\ (\varepsilon,\sigma_1) = \mathcal{R}[\![e]\!](\phi,\varphi)(\rho,\sigma)$$
$$\mathbf{and}\ (l,\sigma_2) = \mathit{allocation}(\sigma_1)$$
$$\mathbf{and}\ \rho_1 = \mathit{overlay}(\mathit{binding}(\nu,l),\rho)$$
$$\mathbf{and}\ \sigma_3 = \mathit{store}(l,\varepsilon,\sigma_2)$$
$$\mathbf{in}\ \mathcal{E}^*[\![e^*]\!](\phi,\varphi)(\rho_1,\sigma_3)$$
$$\mathcal{E}[\![\texttt{(method (}\nu^*\texttt{)}\ e^*\texttt{)}]\!] =$$
$$\lambda(\phi,\varphi).\lambda(\rho,\sigma).\ \mathbf{let}\ (\varepsilon,\sigma_1) = \mathit{make\text{-}meth}(\nu^*,e^*,\sigma)$$
$$\mathbf{in}\ (\varepsilon,\sigma_1)$$
$$\mathcal{E}[\![\texttt{(if}\ e\ e_1\ e_2\texttt{)}]\!] =$$
$$\lambda(\phi,\varphi).\lambda(\rho,\sigma).\ \mathbf{let}\ (\varepsilon,\sigma_1) = \mathcal{R}[\![e]\!](\phi,\varphi)(\rho,\sigma)$$
$$\mathbf{in}\ \ \mathbf{if}\ \varepsilon\mid_{\mathbf{RV}}\mid_{\mathbf{V}}\mid_{\mathbf{T}}$$
$$\mathbf{then}\ \mathcal{E}[\![e_1]\!](\phi,\varphi)(\rho,\sigma_1)$$
$$\mathbf{else}\ \mathcal{E}[\![e_2]\!](\phi,\varphi)(\rho,\sigma_1)$$
$$\mathbf{endif}$$
$$\mathcal{E}[\![\texttt{(set !}\ \nu\ e\texttt{)}]\!] =$$
$$\lambda(\phi,\varphi).\lambda(\rho,\sigma).\ \mathbf{let}\ (\varepsilon,\sigma_1) = \mathcal{R}[\![e]\!](\phi,\varphi)(\rho,\sigma)$$
$$\mathbf{in}\ (\mathit{unspecified}, \mathit{store}(\rho(\nu)\mid_{\mathbf{Loc}},\varepsilon,\sigma_1))$$

FIG. 7.2 – Fonctions sémantiques pour le langage LIEBERMAN

de messages, un environnement pour conserver les liaisons `let`, une mémoire pour conserver les valeurs des emplacements modifiables. Les algèbres pour les mémoire et les environnements sont très connus (voir [Mos90]). Le *self* et le *super* sont des valeurs du domaine **Behavior** introduit précédemment.

7.3.1 Mémoire

Le seul domaine traité de façon inusitée dans notre sémantique est celui de la mémoire. Tout objet a une identité unique représentée par une valeur du domaine **Oop**. Cet identificateur joue deux rôles dans notre sémantique. D'une part, il doit permettre de retrouver l'objet en mémoire et d'autre part il permet le partage des objets par d'autres objets. On peut croire que l'identificateur d'objet pourrait être modélisé par un pointeur, typiquement représenté par une valeur d'un domaine des valeurs à gauche (*left-values*) qui sert à modéliser la mémoire elle-même comme une valeur d'une domaine **Store** qui sont des fonctions prenant une valeur à gauche et retournant une valeur mémorisable.

Le problème avec cette approche est qu'un identificateur d'objet n'est pas une valeur à gauche mais bien une valeur à droite (*right-value*). Cette précision importante est souvent négligée dans les sémantiques formelles des langages à objets. Il n'est donc pas possible d'utiliser directement un pointeur comme identificateur d'objet. La solution adoptée consiste à

construire les domaines **Oop** et **Loc** (de «*location*», emplacement mémoire) à partir d'un même domaine **Pointer**. Cette solution préserve le caractère de valeur à droite de l'identificateur d'objet, tout en permettant d'accéder facilement à l'emplacement mémoire correspondant.

Par ailleurs, les méthodes également doivent être des valeurs partageables entre différents objets. Elles sont donc placées en mémoire et désignées par un identificateur. Les valeur d'objets mémorisables du domaine sémantique **OmV** sont donc divisées entre les objets du domaine **Object** et les méthodes du domaine **Meth**. Les valeurs mémorisables du domaine **SV** contiennent donc les valeurs à droite (constantes et identificateurs d'objets) et les valeurs d'objets mémorisables.

7.3.2 Modèle dénotationnel des objets

Considérons maintenant le modèle dénotationnel des objets. Outre la sémantique des enveloppeurs, un modèle dénotationnel des champs est nécessaire. Un emplacement qui contiendra la valeur d'un champ est alloué en mémoire, et une correspondance entre le nom d'un champ et l'emplacement mémoire correspondant est obtenu par un dictionnaire, c'est-à-dire une valeur du domaine **Dict** qui est une fonction prenant un symbole (nom de champ) et retournant un emplacement mémoire. La valeur d'un champ peut être soit une constante dans **V** ou un identificateur d'objet dans **Oop**.

L'enveloppeur d'objet est construit par la fonction suivante prenant un dictionnaire Δ et retournant un enveloppeur :

$$
\begin{array}{l}
\mathit{make\text{-}wrapper} = \\
\quad \lambda\Delta.\lambda\phi.\lambda\varphi.\lambda q. \\
\qquad \textbf{case } \mathit{tmp} \;=\; \Delta(q) \textbf{ of} \\
\qquad \mathit{is}\textbf{Loc } ? \Rightarrow \\
\qquad\quad \textbf{case } l \;=\; \mathit{tmp}\,|_{\mathbf{Loc}} \textbf{ of} \\
\qquad\quad \mathit{is}\textbf{O } ? \Rightarrow\; \mathit{in}((\mathbf{T} \to \mathbf{Fun}) + \mathbf{O})(\lambda\tau.\lambda\sigma.\lambda\pi^*.\bot_{(\mathbf{EV}\otimes\mathbf{S})}) \;, \\
\qquad\quad \mathit{is}\textbf{Pointer } ? \Rightarrow \\
\qquad\qquad \mathit{in}((\mathbf{T} \to \mathbf{Fun}) + \mathbf{O})(\lambda\tau.\;\; \textbf{if } \tau \\
\qquad\qquad\qquad\qquad\qquad \textbf{then } \lambda\sigma.\lambda\pi^*.\;\; \textbf{case } \vartheta \;=\; \sigma(l)\,|_{\mathbf{SV}}|_{\mathbf{RV}} \textbf{ of} \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathit{is}\textbf{Oop } ? \Rightarrow \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{case } \alpha \;=\; \sigma(\mathit{in}\textbf{Loc}(\vartheta\,|_{\mathbf{Oop}}|_{\mathbf{Pointer}}))\,|_{\mathbf{SV}}|_{\mathbf{OmV}} \textbf{ of} \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{is}\textbf{Meth } ? \Rightarrow\; \alpha\,|_{\mathbf{Meth}}\,(\phi,\varphi)(\sigma)(\pi^*) \;, \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{is}\textbf{Object } ? \Rightarrow\; (\mathit{in}\mathbf{EV}(\vartheta), \sigma) \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{endcase} \;\;, \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathit{is}\textbf{V } ? \Rightarrow\; (\mathit{in}\mathbf{EV}(\vartheta), \sigma) \\
\qquad\qquad\qquad\qquad\qquad\qquad \textbf{endcase} \\
\qquad\qquad\qquad\qquad\qquad \textbf{else } \lambda\sigma.\lambda\pi^*.(\mathit{unspecified}, \mathit{store}(l, \pi^*\downarrow_1, \sigma)) \\
\qquad\qquad\qquad\qquad\qquad \textbf{endif} \;) \\
\qquad\quad \textbf{endcase} \;\;, \\
\qquad \mathit{is}\textbf{T } ? \Rightarrow\; \mathit{in}((\mathbf{T} \to \mathbf{Fun}) + \mathbf{O})(\bot_{\mathbf{O}}) \\
\qquad \textbf{endcase}
\end{array}
$$

Il faut se rappeler que le rôle d'un enveloppeur est de bien lier le self et le super à l'intérieur de ce qui dénote un objet dans le modèle, en particulier une fonction prenant un symbole (sélecteur d'un message) et retournant la dénotation de l'application d'un champ, qui est elle-même une fonction.

L'objet obtenu à partir de l'enveloppeur précédent va répondre à un sélecteur de la façon

suivante. Si le sélecteur est inconnu, la valeur «*bottom*» est retournée (les choses peuvent mal tourner s'il n'y a pas de champ correspondant au sélecteur ou si, pour une raison quelconque, l'emplacement mémoire enregistrée dans le dictionnaire est invalide). Si le sélecteur correspond à un emplacement mémoire valide, l'objet retourne une fonction $\mathbf{T} \rightarrow \mathbf{Fun}$. Cette fonction implante la mutation des objets ; si la valeur **true** lui est passée, elle retourne une fonction qui applique la valeur du champ, alors que si la valeur **false** lui est passée, elle retourne une fonction mettant à jour la valeur du champ. Dans les deux cas, la fonction f retournée est une valeur du domaine **Fun**. f accepte une mémoire (**S**), et retourne une fonction qui accepte une liste de paramètres réels ($\mathbf{PV}^*$) et qui retourne le résultat du message et une nouvelle valeur de mémoire.

Une fonction f de mise-à-jour d'un champ mémorise son unique paramètre réel (extrait de la liste π^*) dans l'emplacement mémoire correspondant au champ. Une fonction f ne faisant pas la mise-à-jour retourne simplement la valeur du champ si cette valeur est une constante (**V**) ou un identificateur d'objet (**Oop**). Une méthode est dénotée par une valeur du domaine **Meth**, qui est une fonction prenant un self, un super et retournant une fonction f dans **Fun**. Si la valeur d'un champ est une méthode, cette dernière est activée en lui passant le self et le super courants pour obtenir la fonction f. Cette fonction f est elle-même exécutée en lui passant la liste des paramètres réels et la mémoire.

7.3.3 Méthodes

Un valeur représentant une méthode dans le domaine **Meth** est créée par la fonction *make-meth*, qui prend une liste de paramètres formels, une liste d'expressions (le corps de la méthode) et une mémoire. *Make-meth* alloue la nouvelle méthode en mémoire et retourne l'identificateur d'objet correspondant ainsi que la nouvelle mémoire. La fonction *make-meth* est définie comme suit :

$$
\begin{array}{l}
\textit{make-meth} = \\
\quad \lambda(\nu^*, e^*, \sigma).\ \textbf{let}\ \alpha = \textit{in}\mathbf{OmV}(\lambda(\phi,\varphi).\lambda\sigma_1.\lambda\pi^*.\ \ \textbf{let*}\ (l^*, \sigma_2) = \textit{allocations}(\#\pi^*)(\sigma_1) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{and}\ \sigma_3 = \textit{assign}(l^*, \pi^*)(\sigma_2) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{and}\ \rho = \textit{bindings}(\nu^*, l^*) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{in}\ \mathcal{R}^*[\![e^*]\!](\phi,\varphi)(\rho,\sigma_3)\ ) \\
\qquad\qquad\qquad \textbf{in}\ \ \textbf{let}\ (l,\sigma_1) = \textit{allocation}(\sigma) \\
\qquad\qquad\qquad\qquad\ \textbf{in}\ (\textit{in}\mathbf{EV}(\textit{in}\mathbf{RV}(\textit{in}\mathbf{Oop}(l\,|_{\mathbf{Pointer}}))), \textit{store}(l, \textit{in}\mathbf{SV}(\alpha), \sigma_1))
\end{array}
$$

Notez que la valeur représentant la nouvelle méthode est injectée dans le domaine **OmV** et qu'un identificateur d'objets lui est attribué afin de lui allouer un espace en mémoire.

Le lecteur devrait également noter deux points importants concernant les méthodes. Premièrement, les méthodes ne créent pas des fermetures au sens des λ-expressions de la programmation fonctionnelle. Ce choix est dû au fait qu'une méthode ne doit pas être close sur son environnement de création ; elle ne doit qu'accéder aux variables locales des objets auxquels elle s'applique. Si elle était une fermeture, la méthode créée dans le contexte d'un objet `o1` (par une expression `newInitials`) conserverait l'accès aux variables locales à `o1`, bien qu'en définitive appartenant à un objet `o2`. Lors d'une activation subséquente de la méthode sur `o2`, celle-ci pourrait modifier l'état de l'objet `o1`, ce qui irait à l'encontre du principe d'encapsulation des objets. Ainsi, plutôt que de créer une fermeture, nous utilisons des envois de messages à self pour accéder aux champs de données des objets.

Deuxièmement, la représentation des méthodes doit accepter en paramètre le self et le super du contexte d'application de la méthode. Ces deux paramètres ne peuvent être liés dans

$$\mathcal{E} :: \mathit{E}\text{xpressions} \to (\mathbf{B}\text{ehavior} \otimes \mathbf{B}\text{ehavior}) \to (\mathbf{E}\text{nv} \otimes \mathbf{S}) \to (\mathbf{EV} \otimes \mathbf{S})$$

$$\mathcal{E}[\![\texttt{(send } e\ q\ e^*\texttt{)}]\!] =$$
$$\lambda(\phi,\varphi).\lambda(\rho,\sigma).$$
$$\quad \textbf{let*}\ (\varepsilon^*,\sigma_1) = \mathcal{RL}[\![\mathit{permute}(\langle e\rangle \S e^*)]\!](\phi,\varphi)(\rho,\sigma)$$
$$\quad \textbf{and}\ (\varepsilon,{\varepsilon^*}_1) = \mathit{decompose}(\mathit{unpermute}(\varepsilon^*))$$
$$\quad \textbf{in}\ \ \textbf{if}\ q =_{\mathbf{Sym}} \text{`}\texttt{set}$$
$$\qquad \textbf{then}\ \mathit{on}_1(\mathit{stored}(\mathit{in}\mathbf{Loc}(\varepsilon\ |_{\mathbf{RV}}|_{\mathbf{Oop}}|_{\mathbf{Pointer}}), \sigma_1)\ |_{\mathbf{OmV}}|_{\mathbf{Object}})$$
$$\qquad\qquad ({\varepsilon^*}_1 \downarrow_1 |_{\mathbf{RV}}|_{\mathbf{V}}|_{\mathbf{Sym}})\ |_{(\mathbf{T}\to\mathbf{Fun})}\ (\mathbf{false})(\sigma_1)({\varepsilon^*}_1 \dagger 1)$$
$$\qquad \textbf{else}\ \mathit{on}_1(\mathit{stored}(\mathit{in}\mathbf{Loc}(\varepsilon\ |_{\mathbf{RV}}|_{\mathbf{Oop}}|_{\mathbf{Pointer}}), \sigma_1)\ |_{\mathbf{OmV}}|_{\mathbf{Object}})(q)\ |_{(\mathbf{T}\to\mathbf{Fun})}\ (\mathbf{true})(\sigma_1)({\varepsilon^*}_1)$$
$$\qquad \textbf{endif}$$

$$\mathcal{E}[\![\texttt{(self } q\ e^*\texttt{)}]\!] =$$
$$\lambda(\phi,\varphi).\lambda(\rho,\sigma).\ \ \textbf{let*}\ (\varepsilon^*,\sigma_1) = \mathcal{RL}[\![\mathit{permute}(e^*)]\!](\phi,\varphi)(\rho,\sigma)$$
$$\qquad\qquad\qquad\quad \textbf{and}\ {\varepsilon^*}_1 = \mathit{unpermute}(\varepsilon^*)$$
$$\qquad\qquad\qquad\quad \textbf{in}\ \ \textbf{if}\ q =_{\mathbf{Sym}} \text{`}\texttt{set}$$
$$\qquad\qquad\qquad\qquad \textbf{then}\ \phi({\varepsilon^*}_1 \downarrow_1 |_{\mathbf{RV}}|_{\mathbf{V}}|_{\mathbf{Sym}})\ |_{(\mathbf{T}\to\mathbf{Fun})}\ (\mathbf{false})(\sigma_1)({\varepsilon^*}_1 \dagger 1)$$
$$\qquad\qquad\qquad\qquad \textbf{else}\ \phi(q)\ |_{(\mathbf{T}\to\mathbf{Fun})}\ (\mathbf{true})(\sigma_1)({\varepsilon^*}_1)$$
$$\qquad\qquad\qquad\qquad \textbf{endif}$$

$$\mathcal{E}[\![\texttt{(super } q\ e^*\texttt{)}]\!] =$$
$$\lambda(\phi,\varphi).\lambda(\rho,\sigma).\ \ \textbf{let*}\ (\varepsilon^*,\sigma_1) = \mathcal{RL}[\![\mathit{permute}(e^*)]\!](\phi,\varphi)(\rho,\sigma)$$
$$\qquad\qquad\qquad\quad \textbf{and}\ {\varepsilon^*}_1 = \mathit{unpermute}(\varepsilon^*)$$
$$\qquad\qquad\qquad\quad \textbf{in}\ \ \textbf{if}\ q =_{\mathbf{Sym}} \text{`}\texttt{set}$$
$$\qquad\qquad\qquad\qquad \textbf{then}\ \varphi({\varepsilon^*}_1 \downarrow_1 |_{\mathbf{RV}}|_{\mathbf{V}}|_{\mathbf{Sym}})\ |_{(\mathbf{T}\to\mathbf{Fun})}\ (\mathbf{false})(\sigma_1)({\varepsilon^*}_1 \dagger 1)$$
$$\qquad\qquad\qquad\qquad \textbf{else}\ \varphi(q)\ |_{(\mathbf{T}\to\mathbf{Fun})}\ (\mathbf{true})(\sigma_1)({\varepsilon^*}_1)$$
$$\qquad\qquad\qquad\qquad \textbf{endif}$$

$$\mathcal{E}[\![\texttt{(newInitials } e\ \texttt{(}q^*\texttt{)}\ \texttt{(}e^*\texttt{))}]\!] =$$
$$\lambda(\phi,\varphi).\lambda(\rho,\sigma).\ \ \textbf{let*}\ (\varepsilon^*,\sigma_1) = \mathcal{RL}[\![\mathit{permute}(\langle e\rangle \S e^*)]\!](\phi,\varphi)(\rho,\sigma)$$
$$\qquad\qquad\qquad\quad \textbf{and}\ (\varepsilon,{\varepsilon^*}_1) = \mathit{decompose}(\mathit{unpermute}(\varepsilon^*))$$
$$\qquad\qquad\qquad\quad \textbf{in}\ \mathit{allocate\text{-}new\text{-}object}(\varepsilon, q^*, {\varepsilon^*}_1, \sigma_1)$$

$$\mathcal{E}[\![\texttt{(clone } e\texttt{)}]\!] =$$
$$\lambda(\phi,\varphi).\lambda(\rho,\sigma).\ \ \textbf{let}\ (\varepsilon,\sigma_1) = \mathcal{R}[\![e]\!](\phi,\varphi)(\rho,\sigma)$$
$$\qquad\qquad\qquad\quad \textbf{in}\ \mathit{clone}(\varepsilon,\sigma_1)$$

$$\mathcal{E}[\![\texttt{(root)}]\!] = \lambda(\phi,\varphi).\lambda(\rho,\sigma).(\mathit{in}\mathbf{EV}(\mathit{in}\mathbf{RV}(\mathit{root\text{-}oop})),\sigma)$$

FIG. 7.3 – Fonctions sémantiques pour le langage LIEBERMAN (suite)

la méthode elle-même puisque cette dernière peut être partagée entre plusieurs objets. De plus, en ce qui concerne le paramètre self, il doit rester ouvert puisque la méthode peut être appliquée lors d'un appel à son détenteur mais aussi lors d'un appel à l'un de ses fils ; nous y reviendrons plus loin.

7.3.4 Envoi de messages

La sémantique d'une expression `send` est la suivante. Premièrement, les arguments et le receveur du message sont évalués. Parce que l'ordre de leur évaluation ne devrait pas être important (non-spécifié), nous appliquons une technique proposée par la sémantique de Scheme [CR91], qui consiste à appliquer des permutations arbitraires *permute* et *unpermute* (qui doivent être

inverses l'une de l'autre) à l'ordre des arguments et du receveur avant et après leur évaluation. Le receveur doit s'évaluer à un identificateur d'objet, qui est ensuite utilisé pour retrouver l'objet correspondant en mémoire. Si le sélecteur du message est le symbole `set`, alors il s'agit d'un message de mutation, auquel cas le nom du champ à modifier est donné par le premier argument. Nous passons alors le symbole auquel s'évalue cet argument à l'objet receveur, puis on passe la valeur `false` pour obtenir une fonction qui va réaliser la mutation desirée du champ en mémoire, comme cela est discuté plus haut. Si le sélecteur n'est pas le symbole `set`, il est directement passé à l'objet, suivi de la valeur `true` pour obtenir une fonction qui va soit retourner la valeur du champ ou appliquer la méthode que ce dernier contiendrait alors. La fonction de valuation est donnée à la figure 7.3.

Les fonctions de valuation pour les expressions `self` et `super` sont similaires, à ceci près que plutôt d'accéder à la mémoire pour trouver l'objet receveur, elles utilisent directement l'argument self ou super comme receveur (voir la figure 7.3).

Dans la fonction de valuation pour les expressions `send`, le lecteur attentif aura peut-être noté que nous projetons d'abord l'objet obtenu de la mémoire sur sa première composante avant de lui passer le sélecteur. En fait, l'objet en mémoire est un enregistrement (*record*) contenant un objet (une valeur du domaine **Behavior**) dans sa première composante. Il contient également le générateur de l'objet, la liste des noms de ses champs, la liste des emplacements mémoire correspondants et le générateur de son parent. Ainsi, un objet en mémoire est une valeur du domaine :

$$\mathbf{Object} = \mathbf{Behavior} \times \mathbf{Generator} \times \mathbf{Sym}^* \times \mathbf{Loc}^* \times \mathbf{Generator}$$

Cette représentation est rendue nécessaire par le fait qu'un objet ne doit pas seulement répondre à des messages. Il doit pouvoir être créés des clones, ce qui demande d'avoir les noms et les valeurs courantes de ses champs et d'utiliser le générateur de son parent pour créer un frère. Concrètement, la fonction de valuation pour les expressions `newInitials` (voir figure 7.3) évalue d'abord les valeurs de champs et l'objet parent avant d'appeler la fonction *allocate-new-object* pour créer le nouvel objet :

$$\begin{array}{l}
\textit{allocate-new-object} = \\
\quad \lambda(\varepsilon, q^*, \varepsilon^*, \sigma).\ \ \textbf{let*}\ (b, \Gamma, {q^*}_1, l^*, \Gamma_1) = \sigma(\textit{in}\mathbf{Loc}(\varepsilon\ |_{\mathbf{RV}}|_{\mathbf{Oop}}|_{\mathbf{Pointer}}))\ |_{\mathbf{SV}}|_{\mathbf{OmV}}|_{\mathbf{Object}} \\
\qquad\qquad\qquad\quad \textbf{and}\ (o, \sigma_1) = \textit{create-object}(q^*, \varepsilon^*, \sigma, \Gamma) \\
\qquad\qquad\qquad\quad \textbf{and}\ (l, \sigma_2) = \textit{allocation}(\sigma_1) \\
\qquad\qquad\qquad\quad \textbf{and}\ \sigma_3 = \textit{store}(l, \textit{in}\mathbf{SV}(\textit{in}\mathbf{OmV}(o)), \sigma_2) \\
\qquad\qquad\qquad\quad \textbf{in}\ (\textit{in}\mathbf{EV}(\textit{in}\mathbf{RV}(\textit{in}\mathbf{Oop}(l\ |_{\mathbf{Pointer}}))), \sigma_3)
\end{array}$$

$$\begin{array}{l}
\textit{create-object} = \\
\quad \lambda(q^*, \varepsilon^*, \sigma, \Gamma).\ \ \textbf{let}\ (l^*, \sigma_1) = \textit{allocate-slot-values}(\varepsilon^*)(\sigma) \\
\qquad\qquad\qquad\quad \textbf{in}\ (\ \ \textbf{let}\ \Gamma_1 = \textit{make-wrapper}(\textit{make-object-dict}(q^*)(l^*)) \triangleright \Gamma \\
\qquad\qquad\qquad\qquad\quad \textbf{in}\ (\textbf{fix}(\Gamma_1), \Gamma_1, q^*, l^*, \Gamma)\ , \sigma_1)
\end{array}$$

Allocate-new-object commence par charger de la mémoire l'objet parent et obtient son générateur. Elle appelle ensuite la fonction *create-object*, qui alloue en mémoire les valeurs des champs du nouvel objet, construit son enveloppeur et retourne un objet mémorisable. *Allocate-new-object* requiert alors un emplacement mémoire libre qui est utilisé comme identificateur d'objet et pour ranger le nouvel objet en mémoire. Elle retourne finalement l'identificateur d'objet et la mémoire mise à jour.

La fonction *clone*, appelée par la fonction de valuation des expressions `clone`, commence aussi par charger de la mémoire l'objet à copier. Elle collecte les valeurs courantes des champs

en utilisant la liste de leurs emplacements mémoire, puis elle appelle la fonction *create-object* en lui passant la liste des noms de champs de l'objet cloné, la liste des valeurs courantes de ses champs et le générateur de son parent. Le nouvel objet est alors alloué dans la mémoire :

$$
\begin{array}{l}
\mathit{clone} = \\
\quad \lambda(\varepsilon,\sigma).\ \textbf{let*}\ (b,\Gamma,q^*,l^*,\Gamma_1) = \sigma(\mathit{in}\mathbf{Loc}(\varepsilon\ |_{\mathbf{RV}}|_{\mathbf{Oop}}|_{\mathbf{Pointer}}))\ |_{\mathbf{SV}}|_{\mathbf{OmV}}|_{\mathbf{Object}} \\
\qquad\quad \textbf{and}\ \omega^* = \mathit{collect\text{-}values}(l^*,\sigma) \\
\qquad\quad \textbf{and}\ (o,\sigma_1) = \mathit{create\text{-}object}(q^*,\omega^*,\sigma,\Gamma_1) \\
\qquad\quad \textbf{and}\ (l,\sigma_2) = \mathit{allocation}(\sigma_1) \\
\qquad\quad \textbf{and}\ \sigma_3 = \mathit{store}(l,\mathit{in}\mathbf{SV}(\mathit{in}\mathbf{OmV}(o)),\sigma_1) \\
\qquad\quad \textbf{in}\ (\mathit{in}\mathbf{EV}(\mathit{in}\mathbf{RV}(\mathit{in}\mathbf{Oop}(l\ |_{\mathbf{Pointer}}))),\sigma_3)
\end{array}
$$

$$
\begin{array}{l}
\mathit{collect\text{-}values} = \\
\quad \lambda(l^*,\sigma).\mathbf{fix}(\lambda f.\lambda {l^*}_1.\ \ \textbf{if}\ {l^*}_1 = \langle\rangle \\
\qquad\qquad\qquad\qquad\quad \textbf{then}\ \langle\rangle \\
\qquad\qquad\qquad\qquad\quad \textbf{else}\ \langle\sigma({l^*}_1\downarrow_1)\ |_{\mathbf{SV}}\rangle\S f({l^*}_1\dagger 1) \\
\qquad\qquad\qquad\qquad\quad \textbf{endif}\ \ )(l^*)
\end{array}
$$

7.3.5 Objet initial *root*

Le premier objet dans le système est l'objet *root* dont l'identificateur est lié à la variable *root-oop* et utilisé pour ranger cet objet dans la mémoire initiale du système liée à la variable *initial-om* :

$$\mathit{root\text{-}gen} = \ \lambda\phi.\lambda q.\mathit{in}((\mathbf{T}\to\mathbf{Fun})+\mathbf{O})(\lambda\tau.\lambda\sigma.\lambda\pi^*.\bot_{(\mathbf{EV}\otimes\mathbf{S})})$$

$$
\begin{array}{l}
\mathit{root\text{-}oop} = \\
\quad \textbf{let}\ (l,\sigma) = \mathit{allocation}(\mathit{empty\text{-}store}) \\
\quad \textbf{in}\ \mathit{in}\mathbf{Oop}(l\ |_{\mathbf{Pointer}})
\end{array}
$$

$$
\begin{array}{l}
\mathit{initial\text{-}store} = \\
\quad \textbf{let}\ (l,\sigma) = \mathit{allocation}(\mathit{empty\text{-}store}) \\
\quad \textbf{in}\ \mathit{store}(l,\mathit{in}\mathbf{SV}(\mathit{in}\mathbf{OmV}((\mathbf{fix}(\mathit{root\text{-}gen}),\mathit{root\text{-}gen},\langle\rangle,\langle\rangle,\bot_{\mathbf{Generator}}))),\sigma)
\end{array}
$$

7.3.6 Observations générales

Ceci complète la sémantique dénotationnelle du langage LIEBERMAN. Quelques fonctions sémantiques auxiliaires apparaissent à la figure 7.4 ; elles sont relativement connues et devraient être autodescriptives. Un dernier point important méritant d'être mentionné concerne la représentation des objets dans cette sémantique. La préservation des générateurs d'objets à l'exécution est un aspect très important quoique prévisible[1].

En pratique, cela insiste sur le fait que la pseudo-variable `self` ne peut être liée même dans les prototypes individuels, une importante différence en comparaison avec les objets des langages à classes. Dans les langages à classes, la liaison tardive de `self` est causée par l'application des méthodes définies par une classe à plusieurs instances. À l'intérieur d'un objet spécifique, la valeur de `self` est parfaitement connue et une implantation prête à en payer le prix (en termes d'espace, bien entendu) peut copier les méthodes dans les objets

[1] Ceci a été indépendamment mis en évidence par Steyaert et De Meuter [SM95] dont la discussion est d'ailleurs très intéressante.

individuels auxquels elles s'appliquent. Dans chaque copie, toutes les références à `self` peuvent alors être liées à l'objet lui-même. Bien que cela ne soit pas la façon standard d'implanter les références à `self`, cette observation a été utilisée plusieurs fois dans le passé pour implanter des optimisations (jetez un coup d'œil aux compilations multiples de Self [CUL89]). Les prototypes ne possèdent pas cette propriété du `self` connu.

Même si elles sont attachées aux prototypes, les méthodes conservent le même genre de liaison tardive de leurs références à `self` que dans les langages à classes parce qu'elles doivent également s'appliquer aux objets descendants de l'objet auquel elles appartiennent. Il demeure possible de les copier pour lier le `self`, mais le prototype en arrive alors à avoir deux copies de la méthode, une avec un `self` ouvert pour les descendants (potentiellement à venir) et une avec le `self` lié (ce qui est l'approche de Self à la compilation multiple). Cependant, l'objet lui-même n'a pas de `self` lié. Le `self` est plutôt lié message par message, lors de la réception. Ce phénomène est en réalité la manifestation dans la sémantique formelle de ces langages du problème du moi, évoqué au chapitre précédent.

7.4 Travaux connexes

Comme nous l'avons mentionné, le modèle dénotationnel des objets présenté ici est fortement influencé par les travaux de Cook et Palsberg [CP89], desquels nous avons retenu les techniques des enveloppeurs et des générateurs. Nous avons également trouvé une intéressante source d'inspiration dans la thèse de Mario Wolczko sur la sémantique de Smalltalk [Wol88], de laquelle nous avons adopté la structure générale des domaines sémantiques pour la mémoire d'objet. Par contre, Wolczko place les valeurs des variables d'instance directement dans le dictionnaire de l'objet, sans passer par une indirection dans la mémoire. Bien que cette approche soit suffisante pour modéliser les objets, elle est plus abstraite que la nôtre et donc plus éloigné d'une implantation réelle. D'autres travaux touchant la sémantique dénotationnelle des langages à objets sont ceux de Bracha et Cook [BC90], Cook, Hill et Canning [CHC90], Jagannathan et Agha [JA92], Kamin [Kam88], et finalement Reddy [Red88].

$$
\begin{array}{l}
\mathcal{LL} :: \mathit{C}\text{onstantes}^* \to \mathbf{V}^* \\
\mathcal{LL}[\![k^*]\!] = \ \textbf{if}\ k^* = \langle\rangle\ \textbf{then}\ \langle\rangle\ \textbf{else}\ \langle \mathcal{L}[\![k^* \downarrow_1]\!]\rangle \S \mathcal{LL}[\![k^* \dagger 1]\!]\ \textbf{endif} \\
\mathcal{R} :: \mathit{E}\text{xpressions} \to (\mathbf{B}\text{ehavior} \otimes \mathbf{B}\text{ehavior}) \to (\mathbf{E}\text{nv} \otimes \mathbf{S}) \to (\mathbf{E}\text{V} \otimes \mathbf{S}) \\
\mathcal{R}[\![e]\!] = \\
\quad \lambda(\phi,\varphi).\lambda(\rho,\sigma).\ \ \textbf{let}\ (\varepsilon,\sigma_1) = \mathcal{E}[\![e]\!](\phi,\varphi)(\rho,\sigma) \\
\qquad\qquad\qquad\qquad \textbf{in}\ \ \textbf{if}\ \mathit{is}\mathbf{LV}\ ?(\varepsilon) \\
\qquad\qquad\qquad\qquad\quad \textbf{then}\ (\mathit{stored}(\varepsilon\ |_{\mathbf{LV}}|_{\mathbf{Loc}}, \sigma_1), \sigma_1) \\
\qquad\qquad\qquad\qquad\quad \textbf{else}\ (\varepsilon, \sigma_1) \\
\qquad\qquad\qquad\qquad\quad \textbf{endif} \\
\mathcal{R}^* :: \mathit{E}\text{xpressions}^* \to (\mathbf{B}\text{ehavior} \otimes \mathbf{B}\text{ehavior}) \to (\mathbf{E}\text{nv} \otimes \mathbf{S}) \to (\mathbf{E}\text{V} \otimes \mathbf{S}) \\
\mathcal{R}^*[\![e^*]\!] = \\
\quad \lambda(\phi,\varphi).\lambda(\rho,\sigma).\ \ \textbf{if}\ \#e^* = 1 \\
\qquad\qquad\qquad\qquad \textbf{then}\ \mathcal{R}[\![e^* \downarrow_1]\!](\phi,\varphi)(\rho,\sigma) \\
\qquad\qquad\qquad\qquad \textbf{else}\ \ \textbf{let}\ (\varepsilon,\sigma_1) = \mathcal{R}[\![e^* \downarrow_1]\!](\phi,\varphi)(\rho,\sigma) \\
\qquad\qquad\qquad\qquad\quad\ \ \textbf{in}\ \mathcal{R}^*[\![e^* \dagger 1]\!](\phi,\varphi)(\rho,\sigma_1) \\
\qquad\qquad\qquad\qquad \textbf{endif} \\
\mathcal{RL} :: \mathit{E}\text{xpressions}^* \to (\mathbf{B}\text{ehavior} \otimes \mathbf{B}\text{ehavior}) \to (\mathbf{E}\text{nv} \otimes \mathbf{S}) \to (\mathbf{E}\text{V}^* \otimes \mathbf{S}) \\
\mathcal{RL}[\![e^*]\!] = \\
\quad \lambda(\phi,\varphi).\lambda(\rho,\sigma).\ \ \textbf{if}\ \#e^* = 0 \\
\qquad\qquad\qquad\qquad \textbf{then}\ (\langle\rangle, \sigma) \\
\qquad\qquad\qquad\qquad \textbf{else}\ \ \textbf{let*}\ (\varepsilon,\sigma_1) = \mathcal{R}[\![e^* \downarrow_1]\!](\phi,\varphi)(\rho,\sigma) \\
\qquad\qquad\qquad\qquad\quad\ \ \textbf{and}\ (\varepsilon^*,\sigma_2) = \mathcal{RL}[\![e^* \dagger 1]\!](\phi,\varphi)(\rho,\sigma_1) \\
\qquad\qquad\qquad\qquad\quad\ \ \textbf{in}\ (\langle\varepsilon\rangle \S \varepsilon^*, \sigma_2) \\
\qquad\qquad\qquad\qquad \textbf{endif} \\
\mathcal{E}^* :: \mathit{E}\text{xpressions}^* \to (\mathbf{B}\text{ehavior} \otimes \mathbf{B}\text{ehavior}) \to (\mathbf{E}\text{nv} \otimes \mathbf{S}) \to (\mathbf{E}\text{V} \otimes \mathbf{S}) \\
\mathcal{E}^*[\![e^*]\!] = \\
\quad \lambda(\phi,\varphi).\lambda(\rho,\sigma).\ \ \textbf{if}\ \#e^* = 1 \\
\qquad\qquad\qquad\qquad \textbf{then}\ \mathcal{E}[\![e^* \downarrow_1]\!](\phi,\varphi)(\rho,\sigma) \\
\qquad\qquad\qquad\qquad \textbf{else}\ \ \textbf{let}\ (\varepsilon,\sigma_1) = \mathcal{E}[\![e^* \downarrow_1]\!](\phi,\varphi)(\rho,\sigma) \\
\qquad\qquad\qquad\qquad\quad\ \ \textbf{in}\ \mathcal{E}^*[\![e^* \dagger 1]\!](\phi,\varphi)(\rho,\sigma_1) \\
\qquad\qquad\qquad\qquad \textbf{endif}
\end{array}
$$

FIG. 7.4 – Fonctions sémantiques auxiliaires pour le langage LIEBERMAN

Bibliographie

[Bar81] Henk P. Barendregt. *The Lambda Calculus : Its Syntax and Semantics*. North-Holland, Amsterdam, 1981.

[BC90] Gilad Bracha et William Cook. Mixin-based Inheritance. *Proceedings of OOPSLA/ECOOP'90, ACM Sigplan Notices*, 25(10) :303–311, October 1990.

[Cha96] J. Chazarain. *Programmer avec Scheme — de la pratique à la théorie*. International Thompson Publishing, 1996.

[CHC90] W.R. Cook, W.L. Hill, et P.S. Canning. Inheritance is Not Subtyping. In *Proceedings of the ACM Symposium on Principles of Programming Languages '90*, pages 125–135. ACM Press, January 1990.

[CP89] W. Cook et J. Palsberg. A Denotational Semantics of Inheritance and its Correctness. *Proceedings of OOPSLA'89, ACM Sigplan Notices*, 24(10) :433–443, October 1989.

[CR91] W. Clinger et J. Rees (édité par) . *Revised^4 Report on the Algorithmic Language Scheme*, November 1991.

[CUL89] C. Chambers, D. Ungar, et E. Lee. An Efficient Implementation of Self, a Dynamically-typed Object-Oriented Language Based on Prototypes. *Proceedings of OOPSLA'89, ACM Sigplan Notices*, 24(10) :49–70, October 1989.

[FWH92] D.P. Friedman, M. Wand, et C.T. Haynes. *Essentials of Programming Languages*. MIT Press and McGraw-Hill, 1992.

[Han94] C. Hankin. *Lambda Calculi — A guide for computer scientists*. Oxford Science Publications, 1994.

[Hen91] A. V. Hense. Wrapper semantics of an object oriented programming language with state. In *International Conference on Theoretical Aspects of Computer Software, TACS'91*, pages 548–568, September 1991.

[JA92] S. Jagannathan et G. Agha. A Reflective Model of Inheritance. In *Proceedings of ECOOP'92, Lecture Notes in Computer Science*, volume 615, pages 350–371. Springer-Verlag, July 1992.

[Kam88] S. Kamin. Inheritance in Smalltalk-80 : A denotational definition. In *Proceedings of the ACM Symposium on Principles of Programming Languages '88*, pages 80–87. ACM Press, January 1988.

[Lal90] R. Lalemant. *Logique, réduction, résolution*. Masson, 1990.

[Lie86] H. Lieberman. Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems. *Proceedings of OOPSLA'86, ACM Sigplan Notices*, 21(11) :214–223, November 1986.

[Mos90] P.D. Mosses. Denotational Semantics. In *Handbook of Theoretical Computer Science*, chap. 11, pages 575–631. Elsevier Science Publishers & MIT Press, 1990.

[Que94] C. Queinnec. *Les Langages Lisp*. InterÉditions, 1994.

[Red88] U.S. Reddy. Objects as Closures : Abstract Semantics of Object-Oriented Languages. In *Proceedings of the ACM Conference on Lisp and Functional Programming '88*, pages 289–297. ACM Press, June 1988.

[Sch86] D.A. Schmidt. *Denotational Semantics : a methodology for language development*. Wm. C. Brown Publishers, 1986.

[Set96] R. Sethi. *Programming Languages – Concepts and constructs*. Addison-Wesley, 2nd édition, 1996.

[SK95] K. Slonneger et B.L. Kurtz. *Formal Syntax and Semantics of Programming Languages — a laboratory based approach*. Addison-Wesley, 1995.

[SM95] P. Steyaert et W. De Meuter. A Marriage of Class- and Object-Based Inheritance Without Unwanted Children. In *Proceedings of ECOOP'95*, *Lecture Notes in Computer Science*, volume 952, pages 127–144. Springer-Verlag, August 1995.

[USC+91] D. Ungar, R. Smith, C. Chambers, B.-W. Chang, et U. Hölzle. Special Issue on the Self programming language. *Lisp and Symbolic Computation*, (4), 1991.

[Win93] G. Winskel. *The Formal Semantics of Programming Languages*, *Foundations of Computing Series*. Foundations of Computing Series. MIT Press, 1993.

[Wol88] M. Wolczko. *Semantics of Object-Oriented Languages*. Thèse de doctorat (PhD), University of Manchester, March 1988.