Chapitre 1

Langage

Nous présentons ici, sous un angle formel, les choix effectués dans cette thèse concernant le langage étudié et sa sémantique. Ce langage sera commun, à quelques détails près, aux différents systèmes de types que nous présenterons par la suite. Sa sémantique restera également inchangée dans toute cette thèse.

1.1 Formalisation du langage

Le but de cette thèse est d'étudier différentes techniques de sous-typage au dessus d'un langage à la ML (cf. [CD86]). Ces techniques ont toutes pour but d'étendre le nombre de programmes acceptés par le typeur, mais par des biais différents. Nous verrons par la suite qu'en combinant ces approches « orthogonales », nous pouvons obtenir des systèmes de types plus riches permettant de modéliser, sans ajout de construction de langage, des « concepts » de plus haut niveau comme par exemple les « objets ». Pour permettre ces combinaisons, le langage étudié dans cette thèse varie assez peu en fonction des chapitres, tout comme sa sémantique. Il s'agit d'un ML classique étendu avec des « constructeurs de données » et du « filtrage de motifs ».

1.1.1 Le noyau du langage

Le noyau de notre langage est un simple λ-calcul. Il s'agit d'un langage « à expressions », au sens où tout programme dans notre langage est une expression. L'ensemble $\mathtt{e}$ des expressions est défini par la grammaire suivante :

$$\mathtt{e} ::= \mathtt{x} \mid \lambda\ \mathtt{x}\ .\ \mathtt{e} \mid \mathtt{e}_1\ \mathtt{e}_2$$

On nomme « λ-terme » une expression de ce langage. Dans cette définition, comme en λ-calcul classique, $\mathtt{x}$ représente l'ensemble des variables utilisables dans les programmes, la construction $(\lambda\ \mathtt{x}\ .\ \mathtt{e})$ représente une fonction ayant pour paramètre $\mathtt{x}$ et pour corps $\mathtt{e}$, et la construction $(\mathtt{e}_1\ \mathtt{e}_2)$ représente l'application de la fonction $\mathtt{e}_1$ sur l'argument $\mathtt{e}_2$.

Chaque variable apparaissant dans un λ-terme est alors soit libre, soit liée par λ. Les termes auxquels nous nous intéresserons dans cette thèse seront tous « clos » au sens où ils ne contiendront aucune variable libre.

Le λ-calcul est déjà suffisant pour représenter n'importe quel « calcul ». Il est important de remarquer que l'évaluation d'une expression ne se résume pas à une simple réduction des membres du terme un à un jusqu'à obtenir le résultat comme le serait la réduction d'une expression arithmétique classique. Il est en particulier possible d'encoder en un simple λ-terme un calcul qui boucle infiniment. Un exemple très classique d'un tel terme est :

```
(λ x . x x) (λ x . x x)
```

Il est également possible de définir des constantes et des opérateurs sur ces constantes sous forme de λ-termes. Par exemple, il est possible de définir les constantes entières avec la méthode de Church :

```
0 ≜ λ f x . x
1 ≜ λ f x . f x
2 ≜ λ f x . f (f x)
3 ≜ λ f x . f (f (f x))
...
```

et les opérateurs successeur, addition et multiplication ainsi :

```
succ ≜ λ n f x . f (n f x)
 (+) ≜ λ m n . m succ n
 (×) ≜ λ m n f . m (n f)
```

Le lecteur pourra vérifier que ces opérateurs, appliqués à des λ-termes représentant des entiers comme définis précédemment, vérifient bien la sémantique standard des opérateurs arithmétiques. Ce genre de technique peut servir à définir de nombreux concepts calculatoires et structures de contrôle, et c'est ce qui fait la puissance et la beauté du λ-calcul.

Pour l'expressivité du langage, il serait donc suffisant de se limiter au λ-calcul. Néanmoins, dans le cadre plus concret d'un langage de programmation, tout définir à partir de λ-termes est problématique. La syntaxe du langage peut certes masquer la complexité des λ-termes cachés derrière les valeurs manipulées, mais les performances d'un évaluateur de λ-calcul sont rapidement dépassées par celles d'un calculateur travaillant sur des données représentées dans un format « binaire » plus classique.

De plus, du point de vue de l'analyse statique, une représentation de toutes les valeurs sous forme de λ-termes peut affaiblir les vérifications de la cohérence d'un programme. En effet, plusieurs « concepts » différents peuvent être représentés par le même λ-terme et empêcher la détection de certaines confusions dans le code. À l'inverse, il est intéressant d'associer des « types de base » différents aux différentes classes de constantes et d'enrichir le langage en structures de contrôle pour les distinguer lors du typage des programmes.

1.1.2 Les constantes

Comme expliqué précédemment, plutôt que de définir les constantes sous forme de λ-termes, nous préférons étendre la définition des expressions avec un ensemble `c` de « constantes prédéfinies » :

```
e +:= c
c ::= () | true | false | n | s
n ::= 0 | 1 | -1 | ...
s ::= "" | ...
```

où « `+:=` » désigne l'extension d'une règle de grammaire existante avec une ou plusieurs nouvelles constructions de syntaxe.

L'ensemble des constantes n'est volontairement pas figé dans le contexte théorique de cette thèse. Bien évidemment, une implémentation concrète d'un langage de programmation, pour qu'elle soit pratique à utiliser, définira un ensemble de constantes bien fourni. Cet ensemble contiendra typiquement les booléens, les entiers, les flottants, les caractères, les chaînes de caractères, etc.

1.1.3 Les primitives

Notre langage étant muni de constantes définies autrement que par des λ-termes, il est nécessaire de l'enrichir de primitives permettant de manipuler ces constantes. Ces primitives correspondent aux opérateurs de base que l'on trouve classiquement dans les langages : les opérateurs logiques sur les booléens, les opérateurs arithmétiques classiques sur les nombres entiers et flottants, des opérateurs de manipulation des chaînes, etc. Nous enrichissons donc notre langage avec deux constructions syntaxiques correspondant à l'application de primitives unaires (notées $\mathtt{p}^1$) et binaires (notées $\mathtt{p}^2$) :

$$\mathtt{e} \;+{:=}\; \mathtt{p}^1\ \mathtt{e} \mid \mathtt{p}^2\ \mathtt{e}_1\ \mathtt{e}_2$$

$$\mathtt{p}^1 \;::=\; \mathtt{(not)} \mid \mathtt{(\sim -)} \mid \ldots$$

$$\mathtt{p}^2 \;::=\; \mathtt{(+)} \mid \mathtt{(-)} \mid \ldots$$

où l'opérateur $(\sim -)$ est le moins unaire.

Pour des raisons de lisibilité, on s'autorisera par la suite à utiliser les opérateurs unaires en notation préfixe, et les opérateurs binaires en notation infixe avec les règles de priorité standards. On notera ainsi « $\mathtt{e}_1\ \mathtt{+}\ \mathtt{e}_2$ » plutôt que « $\mathtt{(+)}\ \mathtt{e}_1\ \mathtt{e}_2$ ».

Nous remarquerons que notre définition des primitives ne permet de les utiliser dans une expression que en leur passant immédiatement tous leurs arguments. Certains langages comme OCaml permettent d'utiliser les opérateurs seuls en tant que fonctions. Il est ainsi possible d'écrire « `let f = (+) in ...` » ce qui est équivalent à « `let f = (`λ `x y . x + y) in ...` ». Nous préférons ne pas introduire cette notation dans notre définition des expressions car cela compliquerait inutilement les règles de typage et de sémantique. Une telle notation peut alors simplement être vue comme du sucre syntaxique, et donc expansée lors de l'analyse syntaxique des programmes.

Tout comme pour les constantes, notre langage sera donc toujours « paramétré » par un ensemble de primitives. Pour pouvoir travailler avec cet ensemble inconnu de primitives, il devra être fourni avec deux fonctions δ_1 et δ_2 définissant la sémantique des primitives respectivement unaires et binaires (voir la section 1.2), et une fonction T de typage des constantes et des primitives utilisée dans les systèmes de types (voir la section 2.3). Les fonctions T, δ_1 et δ_2 devront également être liées par une relation de compatibilité pour assurer la validité du typage vis-à-vis de la sémantique (voir la section 2.4.5 du chapitre 2).

1.1.4 Les n-uplets

Il est parfois pratique en programmation de regrouper, dans une même valeur, plusieurs valeurs en créant un couple, un triplet, un quadruplet, etc. De telles constructions sont bien sûr encodables avec de simples λ-termes mais pour les mêmes raisons que précédemment, il est en général préférable d'étendre le langage des expressions avec une construction syntaxique spécifique. Pour simplifier les règles de sémantique et de typage, on préférera ne définir que les 2-uplets (ou « couples ») et encoder les n-uplets pour $n \geqslant 3$ comme une imbrication de couples `(`$\mathtt{e}_1$`, (`$\mathtt{e}_2$`, (`$\mathtt{e}_3$`, ...)))`. Nous n'ajoutons alors que la construction de couple à notre définition des expressions :

$$\mathtt{e} \mathrel{+}= \mathtt{(}\mathtt{e}_1\mathtt{,}\ \mathtt{e}_2\mathtt{)}$$

Pour extraire les valeurs d'un couple, l'approche standard consiste à étendre le filtrage de motifs (défini en section 1.1.9). Par simplicité pour les règles de sémantique et de typage, nous préférons réserver, dans cette thèse, le filtrage de motifs aux constructeurs de données (voir section 1.1.5) et définir à la place les deux primitives d'arité 1, `fst` et `snd` dont la sémantique est respectivement d'extraire le premier et le second élément d'un couple :

$$\mathtt{p}^1 \mathrel{+}= \mathtt{fst} \mid \mathtt{snd}$$

1.1.5 Les constructeurs de données

Les constructeurs de données (aussi appelés « variants ») que nous considérons ici permettent d'accoler une « marque » à une valeur. Cette marque est simplement un nom commençant par une majuscule pour le distinguer des noms de variables.

Ils sont très classiques en programmation fonctionnelle et remplacent en une unique construction les « `union` », « `enum` » et « `struct` » de C. Ils permettent d'encoder des structures de données arborescentes, comme par exemple en OCaml des arbres comportant des chaînes aux noeuds et des entiers aux feuilles :

```
type tree = Leaf of int | Node of string * tree * tree
```

Pour des raisons pratiques, les variants prennent dans certains langages un nombre d'arguments variable (aucun, un ou plusieurs). Pour simplifier la sémantique et le typage de notre langage, nous préférons nous limiter ici à des constructeurs de données à un argument, sachant qu'il est

toujours possible de passer un n-uplet en argument pour simuler un constructeur de données à plusieurs arguments, et de passer la constante `()` en argument pour simuler un constructeur de données sans argument. On étend alors la définition des expressions ainsi :

```
e +:= K e
```

Pour simplifier la syntaxe, nous considérerons par la suite qu'un constructeur seul `K` est une expression valide, simple sucre syntaxique pour `K ()`.

Les constructeurs de données étudiés dans cette thèse ont un statut différent en fonction des chapitres. En premier lieu, c'est-à-dire dans le chapitre 3, les constructeurs de données sont similaires aux variants polymorphes d'OCaml. Ils ne sont donc pas « déclarés » ni associés à un « constructeur de type » comme le sont des variants classiques. En revanche, il est possible de les utiliser directement au milieu du code en indiquant simplement le nom du constructeur et son argument.

Le chapitre 4 ne s'intéresse pas aux constructeurs de données et ils y seront donc ignorés par soucis de simplicité.

Dans le chapitre 5, les constructeurs de données étudiés sont une variante des **`GADT`**. Ils sont déclarés dans le code via une construction de syntaxe spécifique leur attribuant des contraintes de sous-typage. La définition exacte des **`GADT`** nécessite quelques prérequis sur le système de types et ne sera donc donnée que lorsque cela sera possible, au début du chapitre 5.

1.1.6 La construction « `let x = e`$_1$ `in e`$_2$ »

Il est courant d'ajouter une construction `let` aux expressions :

```
e +:= let x = e1 in e2
```

La sémantique précise de cette construction est définie dans la section 1.2. Intuitivement, pour évaluer une telle expression en appel par valeur, on commence par évaluer `e`$_1$ et mettre la valeur obtenue « de côté » en l'associant à la variable `x`. On évalue ensuite `e`$_2$ et à chaque fois que l'on y rencontre la variable `x` (et que `x` n'a pas été redéfinie localement par un autre `let x` ou un λ `x`), on l'évalue en la valeur que l'on avait mise de côté comme résultat de l'évaluation de `e`$_1$. La variable `x` peut apparaître plusieurs fois dans `e`$_2$ et la valeur obtenue par évaluation de `e`$_1$ sera alors répliquée autant de fois que nécessaire.

En réalité, la sémantique de cette construction est exactement la même que celle de `((`λ `x . e`$_2$`) e`$_1$`)`. Il y a néanmoins plusieurs intérêts à l'ajouter au λ-calcul. En premier lieu, d'un point de vue « conception de langage », il est en général plus intuitif pour le programmeur d'écrire `(let x = e`$_1$ `in e`$_2$`)` que `((`λ `x . e`$_2$`) e`$_1$`)`. En effet, `e`$_1$ est évaluée avant `e`$_2$ et il est donc plus naturel de l'écrire avant dans le code. De plus, cette construction correspond à la définition d'une « variable locale », commune à la quasi-totalité des langages de programmation.

L'autre intérêt de cette construction concerne le typage à la Damas/Milner (cf. [DM82]). Un traitement particulier, appelé « généralisation » est habituellement effectué en ML sur le typage de

cette construction `let`. Avec cette technique classique de « généralisation », une expression de la forme `(let x = e1 in e2)` est acceptée dans des cas où son équivalent sémantique `((λ x . e2) e1)` ne l'est pas.

Néanmoins, l'intégralité du chapitre 4 est consacré à une autre approche de la généralisation (cf. [DM82]), qui rend en particulier le typage de l'expression `(let x = e1 in e2)` équivalente au typage de `((λ x . e2) e1)`. Dans ce chapitre, l'expression `(let x = e1 in e2)` pourra alors être vue comme du sucre syntaxique sur `((λ x . e2) e1)`.

1.1.7 Le `let` récursif

La construction `let rec` ne sera ajoutée que dans le chapitre 5 concernant les **`GADT`**, car ce n'est que dans ce chapitre qu'elle aura un réel intérêt. Elle sera alors munie d'une annotation de types pour gérer la récursion polymorphe sur un **`GADT`**.

Dans le reste de la thèse, la récursion pourra être encodée simplement via l'utilisation d'un combinateur de point fixe comme :

$$\texttt{fix} \triangleq \lambda\ \texttt{f . ((}\lambda\ \texttt{x . f (}\lambda\ \texttt{v . x x v)) (}\lambda\ \texttt{x . f (}\lambda\ \texttt{v . x x v)))}$$

Nos systèmes de types autorisant les types récursifs, l'utilisation d'un tel λ-terme dans une expression ne posera pas de problème de typage, sauf en cas de récursion polymorphe bien entendu.

1.1.8 La conditionnelle

La conditionnelle comme les autres structures de contrôle peuvent déjà êtres encodées en λ-calcul. Nous préférons ajouter une construction spécifique au langage :

$$\texttt{e +:= if } e_1 \texttt{ then } e_2 \texttt{ else } e_3$$

Ainsi, lorsque l'évaluation de $\texttt{e}_1$ donne la constante `true`, l'évaluation de toute l'expression se réduit à évaluer $\texttt{e}_2$. A contrario, si $\texttt{e}_1$ s'évalue en la constante `false`, seule $\texttt{e}_3$ est évaluée. Si $\texttt{e}_1$ s'évalue en une valeur différente de `true` et de `false`, l'évaluation s'arrête sur une erreur.

Cette construction suppose bien entendu la présence des booléens dans l'ensemble des constantes. La conditionnelle est néanmoins une construction facultative dans un langage muni d'une construction de filtrage de motifs. Il aurait été possible d'encoder les booléens `true` et `false` par les variants `TRUE` et `FALSE`, et de simuler la construction `if-then-else` grâce à un filtrage de motifs défini en section 1.1.9 :

$$\texttt{match e with TRUE} \rightarrow \texttt{e}_1 \parallel \texttt{FALSE} \rightarrow \texttt{e}_2$$

L'ajout de la construction `if-then-else` dans le langage étudié dans cette thèse n'est donc qu'à but pédagogique. Il s'agit en réalité de la seule construction de langage non-triviale permettant de mettre en évidence le fonctionnement des règles de sémantique et de typage.

1.1.9 Le filtrage de motifs

Le « filtrage de motifs » est une construction syntaxique classique permettant de discriminer les valeurs du langage (comme les entiers, les chaînes de caractères, etc.), de « déconstruire » des structures de données (comme les listes, les tableaux, les n-uplets, les enregistrements, les variants, etc.) et d'en extraire les membres s'il y a lieu. Dans cette thèse, pour simplifier, nous ne nous intéresserons qu'à une forme restreinte du filtrage de motifs sur les variants.

Nous étendons donc la définition des expressions avec les deux constructions suivantes :

$$\texttt{e} \mathrel{+}= \texttt{match e with } \texttt{K}_1\ \texttt{x}_1 \to \texttt{e}_1 \parallel \ldots \parallel \texttt{K}_n\ \texttt{x}_n \to \texttt{e}_n$$

$$\texttt{e} \mathrel{+}= \texttt{match e with } \texttt{K}_1\ \texttt{x}_1 \to \texttt{e}_1 \parallel \ldots \parallel \texttt{K}_n\ \texttt{x}_n \to \texttt{e}_n \parallel \texttt{x}_d \to \texttt{e}_d$$

Ces deux constructions comprennent une expression `e` et un ensemble de n « cas » (avec $n \geqslant 1$). Chaque cas est composé d'un motif de la forme $\texttt{K}_i\ \texttt{x}_i$ et d'une expression $\texttt{e}_i$ dans laquelle peut apparaître $\texttt{x}_i$. Tous les $\texttt{K}_i$ d'un même filtrage sont supposés deux à deux distincts, par construction.

La sémantique précise de ces constructions de type `match` est définie dans la section 1.2. Intuitivement, pour évaluer une telle expression, on commence par évaluer l'expression `e`. Si la valeur obtenue est un variant dont le tag est l'un des $\texttt{K}_i$ mentionné dans le motif de l'un des cas, on évalue le $\texttt{e}_i$ correspondant en y associant la variable $\texttt{x}_i$ à l'argument du $\texttt{K}_i$ comme lors de l'évaluation d'un `let`. Sinon, si la construction possède le cas par défaut $\texttt{x}_d \to \texttt{e}_d$, on évalue $\texttt{e}_d$ en y associant $\texttt{x}_d$ à la valeur de `e`, sinon l'évaluation s'arrête sur une erreur.

Pour des raisons de simplicité, nous avons volontairement restreint la construction `match` définie ici au filtrage des variants. Ceci ne limite pas vraiment les possibilités offertes par le langage. En effet, les listes peuvent être encodées grâce à des variants, et le filtrage sur des entiers, chaînes de caractères, etc., peut s'encoder grâce à des `if` imbriqués même s'il est souvent plus lisible d'utiliser un `match` lorsque le nombre de cas est supérieur à 3.

Nous n'avons pas non plus défini le filtrage de motifs imbriqués, c'est-à-dire la possibilité d'écrire des expressions comme :

$$\texttt{match e with } \texttt{K}_1\ (\texttt{K}_2\ \texttt{x}) \to \texttt{e}_2 \parallel \texttt{K}_1\ (\texttt{K}_3\ \texttt{x}) \to \texttt{e}_3 \parallel \texttt{K}_4\ \texttt{y} \to \texttt{e}_4$$

En effet, de tels motifs imbriqués peuvent s'expanser avant le typage en filtrages imbriqués. Pour notre exemple, cela donnerait :

$$\texttt{match e with } \texttt{K}_1\ \texttt{x} \to (\texttt{match x with } \texttt{K}_2\ \texttt{x} \to \texttt{e}_2 \parallel \texttt{K}_3\ \texttt{x} \to \texttt{e}_3) \parallel \texttt{K}_4\ \texttt{y} \to \texttt{e}_4$$

Enfin, nous n'avons pas permis l'utilisation de filtres disjonctifs (aussi nommés « or-patterns ») dans les motifs, c'est-à-dire la possibilité de mentionner, dans un même filtre, la disjonction entre plusieurs constructeurs de données. Avec une telle construction, on pourrait écrire :

$$\texttt{match e with } (\texttt{K}_1\ \texttt{x} \parallel \texttt{K}_2\ \texttt{x}) \to \texttt{e}_{12}$$

signifiant qu'on évalue $\texttt{e}_{12}$ à la fois lorsqu'un $\texttt{K}_1$ ou un $\texttt{K}_2$ est obtenue à l'évaluation de `e`. Les systèmes de types classiques gèrent en général les filtres disjonctifs en interne directement car

ils sont plus subtils à expanser pour obtenir une expression sans filtre disjonctif. Une technique simple d'expansion qui fonctionnerait du point de vue du typage serait de répliquer $\mathtt{e}_{12}$ dans autant de cas que nécessaire ainsi :

$$\mathtt{match\ e\ with\ K_1\ x} \rightarrow \mathtt{e}_{12} \parallel \mathtt{K_2\ x} \rightarrow \mathtt{e}_{12}$$

Cette réplication de $\mathtt{e}_{12}$ risquerait néanmoins de provoquer un ralentissement de la compilation et une expansion du code généré, notamment si plusieurs filtres disjonctifs sont imbriqués les uns dans les autres. Une autre méthode pour expanser les filtres disjonctifs consiste à capturer l'expression commune aux filtres dans une fonction ainsi :

$$\mathtt{let\ f} = \lambda\ \mathtt{x\ .\ e}_{12}\ \mathtt{in}$$

$$\mathtt{match\ e\ with\ K_1\ x} \rightarrow \mathtt{f\ x} \parallel \mathtt{K_2\ x} \rightarrow \mathtt{f\ x}$$

Une telle expansion du filtrage disjonctif provoquerait néanmoins l'allocation implicite d'une fermeture représentant la fonction `f`, et donc un ralentissement dû aux applications de cette fonction. Cette perte de performance peut cependant être compensée par une passe d'optimisation du code après le typage.

1.1.10 La mutabilité

La mutabilité dans un langage peut être exprimée sous différentes formes, par exemple par la donnée de structures mutables comme les tableaux ou les enregistrements mutables. La technique la plus simple pour ajouter de la mutabilité dans un langage à la ML consiste à uniquement ajouter des « références ». Ces « références » représentent des boîtes stockées en mémoire sur lesquelles sont définies trois opérations :

- La création d'une référence sur une autre valeur
- La lecture du contenu d'une référence
- Le remplacement du contenu d'une référence par une autre valeur

Même si la présence de mutabilité dans un langage est parfois très pratique pour encoder certains algorithmes, son ajout complexifie la syntaxe des règles de sémantique et les preuves de validité de manière orthogonale aux travaux présentés dans cette thèse.

Le langage considéré au long de cette thèse sera donc supposé sans structure de données mutables. Nous présenterons néanmoins à la fin du prochain chapitre comment modifier notre système pour gérer la mutabilité. Nous verrons en particulier que les choix faits dans notre approche du typage simplifieront grandement les liens entre mutabilité et variance.

1.1.11 Résumé

Le langage étudié dans cette thèse peut en résumé être défini par la grammaire suivante :

```
e ::=
    | x | λ x . e | e1 e2
    | c
    | p1 e | p2 e1 e2
    | (e1, e2)
    | K e
    | let x = e1 in e2
    | if e1 then e2 else e3
    | match e with K1 x1 → e1 ‖ ... ‖ Kn xn → en
    | match e with K1 x1 → e1 ‖ ... ‖ Kn xn → en ‖ xd → ed
```

avec les ensembles de constantes et de primitives volontairement laissés ouverts :

```
c  ::= () | true | false | 0 | 1 | -1 | ... | "..." | ...
p1 ::= (not) | (~-) | fst | snd | ...
p2 ::= (&&) | (||) | (+) | (-) | ...
```

Ces grammaires définissent l'ensemble des expressions valides dans notre langage. Nous allons, dans la section suivante, définir formellement comment évaluer ces expressions.

1.2 Sémantique

La sémantique consiste à définir précisément la façon dont les expressions sont évaluées, c'est-à-dire la façon dont on associe une valeur à une expression. Il est important de donner une définition formelle de la sémantique de notre langage puisqu'elle sera utilisée pour définir la notion de « validité du système de types » et la démontrer.

La définition que nous donnons ici de l'évaluation d'une expression ne permet pas d'associer une valeur à toutes les expressions. En effet, l'évaluation d'une expression `e` peut dans certains cas ne pas se terminer et « boucler » indéfiniment, et dans d'autres cas se terminer « sur une erreur ».

Il existe différentes manières de définir la sémantique d'un langage. Une méthode classique consiste à définir une sémantique dite « à environnement ». Cette approche consiste à parcourir l'expression en maintenant un « environnement d'évaluation » représentant l'association entre les variables du programme (introduites par un λ, un `let` ou un `match`) et leur valeur. L'avantage principal de cette approche est d'être similaire à la façon mentale d'imaginer l'évaluation d'une expression, ainsi qu'à son implémentation, que ce soit via un interprète ou l'exécution d'un code compilé. Il est même possible de recourir à une représentation des variables sous la forme d'« indices de de Bruijn » pour se rapprocher plus encore du fonctionnement d'un évaluateur à pile.

L'approche que nous choisissons ici est toutefois très différente, il s'agit d'une sémantique dite « à petits pas, par remplacement, et sans environnement », et a été introduite par Wright et Felleisen [WF92] en 1992. Elle s'éloigne de la façon dont les programmes sont habituellement évalués dans un ordinateur et il ne serait en particulier pas raisonnable, pour des questions de performance, d'implémenter un évaluateur de code de cette manière. Cette technique offre néanmoins la même puissance d'expressivité que les autres approches. Son principal avantage est de simplifier les preuves de validité des systèmes de types.

Le principe de base consiste à ne travailler qu'avec en permanence une unique expression « close », représentant l'état courant de l'évaluateur. Cette expression reste « close » au sens où, à chaque étape de l'évaluation, toutes les variables apparaissant dans l'expression restent liées via un λ, un `let` ou un `match`, ou disparaissent.

Une telle sémantique revient à définir une fonction représentant « un pas d'évaluation », associant une expression « close » à une expression « close ». L'évaluation complète d'une expression reviendra donc à la transformer pas à pas pour former une séquence d'expressions, jusqu'à l'obtention d'une expression que l'on ne peut plus évaluer d'un pas, soit parce qu'elle est devenue « invalide », soit parce qu'on a obtenu l'expression représentant la « valeur résultat ».

1.2.1 Les « valeurs »

Pour définir une sémantique à petit pas, il faut commencer par définir l'ensemble des « valeurs ». Cet ensemble, noté `v`, est ici un sous-ensemble de l'ensemble des expressions `e` défini manuellement

par la grammaire suivante :

$\texttt{v} ::= \lambda\ \texttt{x . e} \mid \texttt{c} \mid (\texttt{v}_1, \texttt{v}_2) \mid \texttt{K v}$

Il peut donc s'agir d'une fonction, d'une constante, d'un couple de valeurs, ou d'un constructeur de données ayant comme argument une valeur. Une valeur ne peut donc pas être une variable seule, une primitive, une application, un `let`, un `if-then-else`, un `match-with`, un couple d'expressions qui ne sont pas des valeurs, ni un constructeur de données appliqué à une expression qui n'est pas une valeur. Néanmoins, de telles expressions peuvent apparaître sous le λ d'une valeur.

Par exemple, la constante `3` et le couple `(K "hello", λ x . x + 1)` sont des valeurs, mais l'expression `K (1 + 2)` ne l'est pas.

Nous remarquerons que la grammaire définissant l'ensemble des valeurs est bien compatible avec celle définissant les expressions, et qu'en particulier, toute valeur est, par construction, une expression. En revanche, toutes les expressions ne sont pas des valeurs.

1.2.2 Un « petit pas » pour l'évaluateur

Un « petit pas d'évaluation », noté « $\longrightarrow$ » représente une réduction d'une expression lorsque cela est possible directement via son constructeur de tête. Il ne faut pas le confondre avec ce que nous nommerons par la suite un « grand pas d'évaluation » qui autorisera la réduction d'une sous-expression et sera noté « $\longmapsto$ ».

Un « petit pas d'évaluation » est en réalité une fonction partielle de l'ensemble `e` des expressions dans lui-même. Il est défini par disjonction des cas de la manière suivante :

Lorsque $\delta_1\ (\texttt{p}^1, \texttt{v})$ est défini :	$\texttt{p}^1\ \texttt{v}$	$\longrightarrow$	$\delta_1\ (\texttt{p}^1, \texttt{v})$
Lorsque $\delta_2\ (\texttt{p}^2, \texttt{v}_1, \texttt{v}_2)$ est défini :	$\texttt{p}^2\ \texttt{v}_1\ \texttt{v}_2$	$\longrightarrow$	$\delta_2\ (\texttt{p}^2, \texttt{v}_1, \texttt{v}_2)$
	$(\lambda\ \texttt{x . e})\ \texttt{v}$	$\longrightarrow$	$\texttt{e}[\texttt{x} \mapsto \texttt{v}]$
	$\texttt{let x = v in e}$	$\longrightarrow$	$\texttt{e}[\texttt{x} \mapsto \texttt{v}]$
	$\texttt{if true then e}_1\ \texttt{else e}_2$	$\longrightarrow$	$\texttt{e}_1$
	$\texttt{if false then e}_1\ \texttt{else e}_2$	$\longrightarrow$	$\texttt{e}_2$
	$\texttt{match K v with}\ldots \parallel \texttt{K x} \to \texttt{e} \parallel \ldots$	$\longrightarrow$	$\texttt{e}[\texttt{x} \mapsto \texttt{v}]$
	$\texttt{match K v with}\ldots \parallel \texttt{K x} \to \texttt{e} \parallel \ldots \parallel \texttt{x}_d \to \texttt{e}_d$	$\longrightarrow$	$\texttt{e}[\texttt{x} \mapsto \texttt{v}]$
Lorsque v n'est pas de la forme $\texttt{K}_i\ \texttt{v}$ avec $1 \leqslant i \leqslant n$:			
	$\texttt{match v with K}_1\ \texttt{x}_1 \to \texttt{e}_1 \parallel \ldots \parallel \texttt{K}_n\ \texttt{x}_n \to \texttt{e}_n \parallel \texttt{x}_d \to \texttt{e}_d$	$\longrightarrow$	$\texttt{e}_d[\texttt{x}_d \mapsto \texttt{v}]$

Cette définition entraîne par exemple que, pour toutes expressions $\texttt{e}_1$ et $\texttt{e}_2$, l'image de l'expression $\texttt{(if true then e}_1\ \texttt{else e}_2\texttt{)}$ par la fonction $(\longrightarrow)$ est $\texttt{e}_1$. Autrement dit, il est possible d'effectuer un petit pas d'évaluation sur l'expression $\texttt{if true then e}_1\ \texttt{else e}_2$, et après ce petit pas, nous obtenons l'expression $\texttt{e}_1$.

Cette définition fait intervenir les fonctions partielles δ_1 et δ_2 mentionnées précédemment en 1.1.3. Pour rappel, le langage étudié dans cette thèse est paramétré par un ensemble de constantes et de primitives, qui est fourni avec deux fonctions d'évaluation δ_1 et δ_2 donnant la sémantique de l'application des opérateurs unaires et binaires à des valeurs.

Typiquement, nous aurons :

- $\delta_2\ (\texttt{(+)}, \texttt{1}, \texttt{2}) = \texttt{3}$
- $\delta_1\ (\texttt{not}, \texttt{true}) = \texttt{false}$
- $\delta_1\ (\texttt{fst}, \texttt{(“hello”, “world”)}) = \texttt{“hello”}$
- etc.

La syntaxe `e[x ↦ v]` représente l'expression `e` dans laquelle toutes les occurrences libres de la variable `x` ont été remplacées par la valeur `v`. La valeur `v` peut ainsi disparaître lorsque `x` n'apparaît pas libre dans `e`, et être répliquée lorsque `x` apparaît plusieurs fois libre dans `e`.

Cette fonction $(\longrightarrow)$ n'est pas définie sur les valeurs (ce qui est normal puisque les valeurs sont des expressions déjà évaluées), mais également sur certaines expressions que l'on souhaiterait être capable d'évaluer, comme par exemple :

```
let n = 1 + 2 in n
```

En effet, l'expression « `1 + 2` » n'est pas une valeur et il est impossible d'appliquer la règle « `let x = v in e` $\longrightarrow$ `e[x ↦ v]` » pour la réduire par $(\longrightarrow)$. Pour évaluer de telles expressions, nous allons maintenant définir les notions de « contexte d'évaluation » et de « réduction de sous-expressions ».

1.2.3 Le contexte d'évaluation

La notion de « contexte d'évaluation » ne doit pas être confondue avec la notion d'« environnement d'évaluation » mentionnée précédemment. Il s'agit ici de définir un ensemble `E` des « expressions possédant un trou noté `[]` » et un opérateur, noté « `_[_]` », permettant de remplir ce trou avec une expression standard `e` (ce que l'on notera `E[e]`) pour obtenir une nouvelle expression sans trou. L'ensemble `E` est à nouveau défini par une grammaire :

```
E ::=
    | []
    | E e | v E
    | p¹ E | p² E e | p² v E
    | (E, e) | (v, E) | K E
    | let x = E in e
    | if E then e₁ else e₂
    | match E with K₁ x₁ → e₁ ‖ ... ‖ Kₙ xₙ → eₙ
    | match E with K₁ x₁ → e₁ ‖ ... ‖ Kₙ xₙ → eₙ ‖ x_d → e_d
```

Cette définition nous impose en particulier que tout contexte d'évaluation possède un et seulement un « trou » noté « `[]` ». Ceci nous permet de définir sans ambiguïté l'opérateur (noté « `_[_]` ») prenant en argument un contexte `E`, une expression `e`, et créant l'expression notée `E[e]` dans laquelle le trou `[]` de `E` a été remplacé par `e`.

Par exemple, si le contexte $\texttt{E}_0$ est défini par :

$$\texttt{E}_0 \triangleq \texttt{let x = 1 + [] in x}$$

et l'expression $\texttt{e}_0$ par :

$$\texttt{e}_0 \triangleq \texttt{if true then 3 else 4}$$

alors, l'expression $\texttt{E}_0\texttt{[e}_0\texttt{]}$ vaut :

$$\texttt{E}_0\texttt{[e}_0\texttt{]} = \texttt{let x = 1 + (if true then 3 else 4) in x}$$

Dans la définition de E, le cas $\texttt{p}^2$ `v E` permet de placer un trou dans le deuxième argument de l'application d'un opérateur binaire lorsque le premier argument est déjà évalué (c'est-à-dire est une valeur). C'est ce qui permet d'évaluer le deuxième argument d'un opérateur binaire avant d'exécuter l'opération. Il est important d'imposer que $\texttt{p}^2$ soit un opérateur binaire dans ce cas car sinon, certaines expressions pourraient avoir plusieurs décompositions de la forme `E[e]`. Par exemple, l'expression e définie par :

$$\texttt{e} \triangleq \texttt{fst ((}\lambda\texttt{ x . x + 1), (}\lambda\texttt{ x . x - 1)) (1 + 2)}$$

pourrait se décomposer en $\texttt{E}_1\texttt{[e}_1\texttt{]}$ avec :

$$\texttt{E}_1 = \texttt{[] (1 + 2)} \quad \text{et} \quad \texttt{e}_1 = \texttt{fst ((}\lambda\texttt{ x . x + 1), (}\lambda\texttt{ x . x - 1))}$$

mais aussi en $\texttt{E}_2\texttt{[e}_2\texttt{]}$ avec :

$$\texttt{E}_2 = \texttt{fst ((}\lambda\texttt{ x . x + 1), (}\lambda\texttt{ x . x - 1)) []} \quad \text{et} \quad \texttt{e}_2 = \texttt{1 + 2}$$

Le fait qu'une même expression puisse se décomposer sous la forme `E[e]` de plusieurs manières posera des problèmes d'ambiguïté dans l'ordre d'évaluation. Il est donc nécessaire que la décomposition sous la forme `E[e]` soit unique, lorsqu'elle existe.

À cause de cette contrainte sur la définition de E, un opérateur ne peut pas être à la fois unaire et binaire. Il s'agit d'une hypothèse sur la définition des fonctions δ_1 et δ_2 : les domaines de primitives sur lesquelles elles s'appliquent doivent être disjoints. Dans le contexte d'un compilateur ou d'un évaluateur, l'opérateur arithmétique standard `(-)` est alors problématique, il doit donc être annoté tôt dans la chaîne de compilation comme étant utilisé sous sa forme binaire ou unaire, typiquement au moment de l'analyse syntaxique.

Le contexte d'évaluation va maintenant nous permettre de définir l'évaluation d'une sous-expression, nécessaire à la définition d'un « grand pas d'évaluation ».

1.2.4 Un « grand pas » pour l'évaluation

Un « grand pas d'évaluation », noté par une flèche à talon « $\longmapsto$ », est défini comme le « passage au contexte » d'un petit pas d'évaluation (noté « $\longrightarrow$ »). Tout comme ($\longrightarrow$), il s'agit d'une fonction partielle définie de l'ensemble des expressions `e` dans lui même. Nous commençons par définir la relation ($\longmapsto$) puis démontrons que c'est une fonction.

Par définition, l'expression $\texttt{e}_2$ est image de $\texttt{e}_1$ par la relation ($\longmapsto$) (ce que l'on notera « $\texttt{e}_1 \longmapsto \texttt{e}_2$ ») s'il existe un contexte d'évaluation `E` et deux expressions $\texttt{e}'_1$ et $\texttt{e}'_2$ tels que les trois conditions suivantes soient vérifiées :

- $\texttt{e}_1 = \texttt{E}[\texttt{e}'_1]$
- $\texttt{e}_2 = \texttt{E}[\texttt{e}'_2]$
- $\texttt{e}'_1 \longrightarrow \texttt{e}'_2$

Pour tout contexte `E` et toutes expressions $\texttt{e}'_1$ et $\texttt{e}'_2$, nous avons donc :

$$\texttt{E}[\texttt{e}'_1] \longmapsto \texttt{E}[\texttt{e}'_2] \quad \Longleftrightarrow \quad \texttt{e}'_1 \longrightarrow \texttt{e}'_2$$

Ainsi, à partir de l'expression $\texttt{e}_1$, un grand pas d'évaluation peut être effectué soit directement par un petit pas d'évaluation (dans le cas où `E = []`), soit en évaluant une sous-expression de $\texttt{e}_1$ par un petit pas. La décomposition de $\texttt{e}_1$ sous la forme $\texttt{E}[\texttt{e}'_1]$ étant unique, il n'y a pas le choix dans la sous-expression de $\texttt{e}_1$ à évaluer par un petit pas. Ceci retire toute ambiguïté sur l'ordre d'évaluation des sous expressions. La définition d'un grand pas d'évaluation que nous avons donnée est donc déterministe, toute expression a au plus une image par ($\longmapsto$), ce qui montre que la relation ($\longmapsto$) est bien une fonction.

1.2.5 Les expressions « bloquées »

La fonction ($\longmapsto$) n'est pas définie pour toutes les expressions. Elle n'est en particulier pas définie sur la partie des expressions que sont les valeurs : aucune valeur ne peut subir un grand pas d'évaluation, le calcul est déjà terminé. Il existe néanmoins d'autres expressions que les valeurs sur lesquelles la fonction ($\longmapsto$) n'est pas définie, elles sont nommées les « expressions bloquées ».

Par définition, une « expression bloquée » est une expression $\texttt{e}_0$ qui n'est pas une valeur et telle qu'il n'existe aucune expression $\texttt{e}_1$ vérifiant $\texttt{e}_0 \longmapsto \texttt{e}_1$.

Par exemple, les expressions suivantes sont bloquées :

- `x`
- `3 “hello”`
- `let x = 1 + true in x`

Être une expression bloquée n'est néanmoins pas équivalent au fait de contenir une sous-expression bloquée. Par exemple, les expressions suivantes ne sont pas bloquées :

- `(1 + 2, match 3 with A -> 4)`
- `if false then fst 4 else 7`

1.2.6 Plusieurs pas d'évaluation

Nous allons maintenant définir une relation entre l'ensemble des expressions `e` et lui même, notée « $\longmapsto\!\!\!\!\!\rightarrow$ », représentant « plusieurs pas d'évaluation ». La relation ($\longmapsto\!\!\!\!\!\rightarrow$) est définie comme la fermeture réflexive transitive de ($\longmapsto$) : par définition, $\texttt{e}_1 \longmapsto\!\!\!\!\!\rightarrow \texttt{e}_2$ si l'une des conditions suivantes est vérifiée :

- $\texttt{e}_1 = \texttt{e}_2$
- $\texttt{e}_1 \longmapsto \texttt{e}_2$
- Il existe une expression intermédiaire `e` telle que $\texttt{e}_1 \longmapsto\!\!\!\!\!\rightarrow \texttt{e}$ et $\texttt{e} \longmapsto \texttt{e}_2$

Il s'agit d'une relation et pas d'une fonction car une expression peut avoir plusieurs images par ($\longmapsto\!\!\!\!\!\rightarrow$). Cette relation nous permet alors de définir la fonction d'évaluation.

1.2.7 La fonction d'évaluation

La fonction d'évaluation, notée `eval`, est une fonction totale et non-calculable (au sens de la théorie de la calculabilité (cf. [S96])) sur l'ensemble des expressions `e`. Elle a pour co-domaine l'ensemble ($\texttt{v} \cup \{ \Uparrow, \textsc{error} \}$) et est définie de la manière suivante :

- `eval(e) = v` si $\texttt{e} \longmapsto\!\!\!\!\!\rightarrow \texttt{v}$ avec `v` une valeur
- `eval(e) = ERROR` s'il existe une expression bloquée `e'` telle que $\texttt{e} \longmapsto\!\!\!\!\!\rightarrow \texttt{e'}$
- `eval(e) =` $\Uparrow$ sinon (lorsque l'évaluation boucle)

Comme les valeurs et les expressions bloquées n'ont pas d'image par ($\longmapsto$), une même expression ne peut pas avoir plusieurs images par `eval`. La définition de la fonction `eval` n'est donc pas ambiguë.

Les expressions bloquées jouent donc un rôle très important : ce sont les expressions qui provoqueraient une erreur à l'exécution si on décidait de les interpréter ou de les compiler et d'exécuter le code produit. Détecter qu'une expression est bloquée n'est pas difficile en soi, toute la difficulté du typage tient à prévoir si l'évaluation d'un code aboutira ou pas à une expression bloquée après un nombre indéfini de pas d'évaluation, autrement dit si `eval` renverra ERROR. C'est une de ces techniques de typage que nous étudions dans le chapitre suivant.