Projet du cours « Compilation »
Jalon 2 : Interprétation de Hopix

version 1.0

1 Syntaxe abstraite

p ::=		**Programmes**
	$\mid \overline{vd}$	Liste de définitions de valeurs
vd ::=		**Définitions de valeur**
	$\mid$ **val** x = e	Valeur simple
	$\mid$ **fun** $\overline{f\ p}$ = e	Fonctions mutuellement récursives
e ::=		**Expressions**
	$\mid n$	Entier litéral
	$\mid c$	Caractère litéral
	$\mid s$	Chaîne de caractères litérale
	$\mid x$	Variable
	$\mid K$ ($\overline{e}$)	Construction d'une valeur étiquetée
	$\mid$ ($\overline{e}$)	n-uplet
	$\mid$ {ℓ = e; $\cdots$; ℓ = e}	Construction d'un enregistrement
	$\mid e.\ell$	Accès à un champ d'enregistrement
	$\mid e$; $\cdots$; e	Séquence
	$\mid vd$; e	Définition locale
	$\mid e\ e$	Application
	$\mid \backslash m$ => e	Fonction anonyme
	$\mid$ **ref** e	Création d'une référence
	$\mid e$:= e	Assignation
	$\mid$! e	Déréférencement
	$\mid$ **match** (e) { $\overline{b}$ }	Filtrage de motif
	$\mid$ **if** (e) **then** { e } **else** { e }	Conditionnelle
	$\mid$ **while** (e) { e }	Boucle non bornée
	$\mid$ **do** { e } **until** (e)	Boucle non bornée non vide
	$\mid$ **for** x **from** (e) **to** (e) **do** { e }	Boucle bornée
b ::=		**Branches**
	$\mid m$ => e	Branche
m ::=		**Motifs**
	$\mid x$	Motif universel liant
	$\mid$ _	Motif universel non liant
	$\mid n$	Entier
	$\mid c$	Caractère
	$\mid s$	Chaîne de caractères
	$\mid K$ ($\overline{m}$)	Motif étiqueté
	$\mid$ ($\overline{m}$)	n-uplet
	$\mid$ {ℓ = m; $\cdots$; ℓ = m}	Enregistrement
	$\mid m$ | m	Disjonction
	$\mid m$ & m	Conjonction

Dans les sections suivantes, lorsque l'on utilisera la syntaxe concrète dans les spécifications, on fera référence implicitement aux arbres de syntaxe abstraite sous-jacents définis par la grammaire ci-dessus. Remarquez que cette grammaire suit la même structure que les arbres de syntaxe abstraite définis dans `HopixAST`, avec comme seule différence l'absence des informations de typage car elles n'interviennent pas dans l'évaluation.

Dans cette grammaire, on a utilisé les *métavariables* suivantes :

— T pour représenter un constructeur de type.
— x ou f pour représenter un identificateur de valeur.
— K pour représenter le nom d'un constructeur de données.
— ℓ pour représenter un identificateur de champ d'enregistrement.
— e pour représenter une expression.
— m pour représenter un motif.
— n pour représenter un litéral de type entier.
— c pour représenter un litéral de type caractère.
— s pour représenter un litéral de type chaîne de caractères.
— b pour représenter une branche d'analyse de motifs.

Par ailleurs, on a noté $\overline{X}$ une liste finie potentiellement vide de X.

2 Interprétation

2.1 Valeurs

Les valeurs du langage sont définies par la grammaire qui suit.

$v ::=$	**Valeurs**
$\mid n$	Entier
$\mid c$	Caractère
$\mid s$	Chaîne de caractères
$\mid K\,(\,\overline{v}\,)$	Valeur étiquetée
$\mid (\,\overline{v}\,)$	n-uplet de valeurs
$\mid \{\ell = v\,; \cdots ; \ell = v\}$	Enregistrement
$\mid (\,\backslash m \Rightarrow e\,)[\sigma]$	Fermeture
$\mid$ prim	Primitive

Environnement d'évaluation Les identificateurs de programme sont associés à des valeurs à l'aide d'un environnement d'évaluation σ. On écrit « $\sigma[x]$ » pour parler de la valeur de x dans l'environnement σ. On écrit « $\sigma + x \mapsto v$ » pour parler de l'environnement σ étendu par l'association entre l'identificateur x et la valeur v. On note $\bullet$ pour l'environnement d'évaluation vide.

Références Les adresses des références sont allouées dynamiquement dans une mémoire ρ. On écrit « $\rho + a \mapsto v$ » pour représenter la mémoire qui étend la mémoire ρ avec une nouvelle adresse a où se trouve stockée la valeur v. On écrit « $\rho[a \leftarrow v]$ » pour représenter la mémoire ρ modifiée seulement à l'adresse a en y stockant la valeur v. Enfin, on écrit « $(a \mapsto v) \in \rho$ » pour indiquer que la valeur v est stockée à l'adresse a de la mémoire ρ. On note $\emptyset$ pour la mémoire vide.

Primitive Les primitives du langage sont définies dans le module `HopixInterpreter`. Ce sont des fonctions OCaml que l'on a rendues accessibles depuis Hopix.

2.2 Évaluation des programmes

Un programme p s'évalue à partir d'une mémoire vide et d'un environnement contenant les primitives du langage en évaluant successivement les définitions de valeurs dans leur ordre d'apparition dans le programme. Comme les types n'ont pas d'existence au moment de l'évaluation, les définitions de type sont ignorées par l'interpréteur.

Pour spécifier précisément ce processus, il faut donc décrire la façon dont les définitions de valeurs s'évaluent : c'est le rôle de la section 2.3. Les expressions sont les termes qui s'évaluent en des valeurs. Leur évaluation est spécifiée dans la section 2.4. Elle s'appuie sur l'évaluation de l'analyse par cas (section 2.5) et l'analyse de motifs (section 2.6).

2.3 Évaluation des définitions

L'évaluation des définitions s'appuie sur le jugement

$$\sigma, \rho \vdash vd \Rightarrow \sigma', \rho'$$

qui se lit « dans l'environnement σ et la mémoire ρ, la définition vd étend σ et σ' et modifie ρ en ρ' ».

Cette partie de la spécification est omise volontairement. Vous devez réfléchir à une spécification raisonnable.

2.4 Évaluation des expressions

Le jugement d'évaluation des expressions s'écrit

$$\sigma, \rho \vdash e \Downarrow v, \rho'$$

ce qui se lit « dans l'environnement d'évaluation σ, l'expression e s'évalue en v et change la mémoire ρ en ρ' ».

Le jugement d'évaluation est défini par les règles suivantes :

E-Int
$$\frac{}{\sigma, \rho \vdash n \Downarrow n, \rho}$$

E-Char
$$\frac{}{\sigma, \rho \vdash c \Downarrow c, \rho}$$

E-String
$$\frac{}{\sigma, \rho \vdash s \Downarrow s, \rho}$$

E-Var
$$\frac{\sigma(x) = v}{\sigma, \rho \vdash x \Downarrow v, \rho}$$

E-Constructor
$$\frac{\forall i \in [1 \dots n], \sigma, \rho_{i-1} \vdash e_i \Downarrow v_i, \rho_i}{\sigma, \rho_0 \vdash K\,(\,e_1, \dots, e_n\,) \Downarrow K\,(\,v_1, \dots, v_n\,), \rho_n}$$

E-Tuple
$$\frac{\forall i \in [1 \dots n], \sigma, \rho_{i-1} \vdash e_i \Downarrow v_i, \rho_i}{\sigma, \rho_0 \vdash (\,e_1, \dots, e_n\,) \Downarrow (\,v_1, \dots, v_n\,), \rho_n}$$

E-Record
$$\frac{\forall i \in [1 \dots n], \sigma, \rho_{i-1} \vdash e_i \Downarrow v_i, \rho_i}{\sigma, \rho \vdash \{\ell_1 = e_1\,; \cdots ; \ell_n = e_n\} \Downarrow \{\ell_1 = v_1\,; \cdots ; \ell_n = v_n\}, \rho_n}$$

E-Field
$$\frac{\sigma, \rho \vdash e \Downarrow \{\ell_1 = v_1\,; \cdots ; \ell_n = v_n\}, \rho' \qquad i \in [1, n]}{\sigma, \rho \vdash e.\ell_i \Downarrow v_i, \rho'}$$

E-Sequence
$$\frac{\forall i \in [1 \dots n], \sigma, \rho_{i-1} \vdash e_i \Downarrow v_i, \rho_i}{\sigma, \rho_0 \vdash e_1\ ;\ \cdots\ ;\ e_n \Downarrow v_n, \rho_n}$$

E-LocalDefinition
$$\frac{\sigma, \rho \vdash vd \Rightarrow \sigma', \rho' \qquad \sigma', \rho' \vdash e \Downarrow v, \rho''}{\sigma, \rho \vdash vd\ ;\ e \Downarrow v, \rho''}$$

E-Application
$$\frac{\sigma, \rho \vdash e_f \Downarrow (\,\backslash m \texttt{ => } e\,)[\sigma_f], \rho' \qquad \sigma, \rho' \vdash e_a \Downarrow v_a, \rho'' \qquad \sigma_f \vdash v_a \sim m \rightsquigarrow \sigma'_f \qquad \sigma'_f, \rho'' \vdash e \Downarrow v, \rho'''}{\sigma, \rho \vdash e_f\ e_a \Downarrow v, \rho'''}$$

E-Allocate
$$\frac{\sigma, \rho \vdash e \Downarrow v, \rho' \qquad a \notin dom(\rho')}{\sigma, \rho \vdash \textbf{ref}\ e \Downarrow a, \rho' + a \mapsto v}$$

E-Dereference
$$\frac{\sigma, \rho \vdash e \Downarrow a, \rho' \qquad (a \mapsto v) \in \rho'}{\sigma, \rho \vdash \texttt{!}\,e \Downarrow v, \rho'}$$

E-Assignment
$$\frac{\sigma, \rho \vdash e \Downarrow a, \rho' \qquad \sigma, \rho' \vdash e' \Downarrow v, \rho''}{\sigma, \rho \vdash e \texttt{ := } e' \Downarrow (\,), \rho''[a \leftarrow v]}$$

E-Match
$$\frac{\sigma, \rho \vdash e \Downarrow v_s, \rho' \qquad \sigma, \rho' \vdash v_s \sim \overline{b} \Downarrow v, \rho''}{\sigma, \rho \vdash \textbf{match}\ (\,e\,)\ \{\,\overline{b}\,\} \Downarrow v, \rho''}$$

E-If-True
$$\frac{\sigma, \rho \vdash e_c \Downarrow \textbf{True}(\,), \rho' \qquad \sigma, \rho' \vdash e_t \Downarrow v, \rho''}{\sigma, \rho \vdash \textbf{if}\ e_c\ \textbf{then}\ e_t\ \textbf{else}\ e_f \Downarrow v, \rho''}$$

E-If-False
$$\frac{\sigma, \rho \vdash e_c \Downarrow \textbf{False}(\,), \rho' \qquad \sigma, \rho' \vdash e_f \Downarrow v, \rho''}{\sigma, \rho \vdash \textbf{if}\ (\,e_c\,)\ \textbf{then}\ \{\,e_t\,\}\ \textbf{else}\ \{\,e_f\,\} \Downarrow v, \rho''}$$

E-While-True
$$\frac{\sigma, \rho \vdash e_b \Downarrow \textbf{True}(\,), \rho' \qquad \sigma, \rho' \vdash e \Downarrow (\,), \rho'' \qquad \sigma, \rho'' \vdash \textbf{while}\ (\,e_b\,)\ \{\,e\,\} \Downarrow (\,), \rho'''}{\sigma, \rho \vdash \textbf{while}\ (\,e_b\,)\ \{\,e\,\} \Downarrow (\,), \rho'''}$$

E-While-False
$$\frac{\sigma, \rho \vdash e_b \Downarrow \textbf{False}(\,), \rho'}{\sigma, \rho \vdash \textbf{while}\ (\,e_b\,)\ \{\,e\,\} \Downarrow (\,), \rho'}$$

E-DoWhile-True
$$\frac{?}{\sigma, \rho \vdash \textbf{do}\ \{\,e\,\}\ \textbf{until}\ (\,e_b\,) \Downarrow ?, ?}$$

E-DoWhile-False
$$\frac{?}{\sigma, \rho \vdash \textbf{do}\ \{\,e\,\}\ \textbf{until}\ (\,e_b\,) \Downarrow ?, ?}$$

E-For-Initialization
$$\frac{\sigma, \rho \vdash e_1 \Downarrow n_1, \rho_1 \qquad \sigma, \rho_1 \vdash e_2 \Downarrow n_2, \rho_2 \qquad \sigma, \rho_2 \vdash \textbf{for}\ x\ \textbf{from}\ (\,n_1\,)\ \textbf{to}\ (\,n_2\,)\ \textbf{do}\ \{\,e\,\} \Downarrow (\,), \rho'}{\sigma, \rho \vdash \textbf{for}\ x\ \textbf{from}\ (\,e_1\,)\ \textbf{to}\ (\,e_2\,)\ \textbf{do}\ \{\,e\,\} \Downarrow (\,), \rho'}$$

E-For-Loop
$$\frac{n_1 \leq n_2 \qquad \sigma[x \mapsto n_1], \rho \vdash e \Downarrow v, \rho' \qquad \sigma, \rho' \vdash \textbf{for}\ x\ \textbf{from}\ (\,n_1 + 1\,)\ \textbf{to}\ (\,n_2\,)\ \textbf{do}\ \{\,e\,\} \Downarrow (\,), \rho''}{\sigma, \rho \vdash \textbf{for}\ x\ \textbf{from}\ (\,n_1\,)\ \textbf{to}\ (\,n_2\,)\ \textbf{do}\ \{\,e\,\} \Downarrow (\,), \rho''}$$

E-For-End
$$\frac{n_1 > n_2}{\sigma, \rho \vdash \textbf{for}\ x\ \textbf{from}\ (\,n_1\,)\ \textbf{to}\ (\,n_2\,)\ \textbf{do}\ \{\,e\,\} \Downarrow (\,), \rho}$$

Notez que les règles traitant du cas des boucles non bornées non vides sont incomplètes. **Cette partie de la spécification est omise volontairement**.

2.5 Analyse par cas

L'analyse par cas d'une valeur v consiste à traiter une liste de cas représentés par des branches $\overline{b}$ de la forme « m `=>` e » en évaluant la première de ces branches dont le motif filtre la valeur v. Ce dernier mécanisme est présenté dans la section suivante.

$$\frac{\sigma \vdash v \sim m \rightsquigarrow \sigma' \qquad \sigma', \rho \vdash e \Downarrow v', \rho'}{\sigma, \rho \vdash v \sim (m \texttt{ => } e)\overline{b} \Downarrow v', \rho'} \qquad \frac{\sigma, \rho \vdash v \not\sim m \qquad \sigma, \rho \vdash v \sim \overline{b} \Downarrow v', \rho'}{\sigma, \rho \vdash v \sim (m \texttt{ => } e)\overline{b} \Downarrow v', \rho'}$$

2.6 Analyse de motifs

L'évaluation des motifs s'appuie sur le jugement

$$\sigma \vdash v \sim m \rightsquigarrow \sigma'$$

qui se lit « dans l'environnement σ, la valeur v est filtrée par le motif m en étendant l'environnement σ en σ' ».

Cette partie de la spécification est omise volontairement. Vous devez réfléchir à une spécification raisonnable.

3 Travail à effectuer

La seconde partie du projet est l'écriture de l'interprète de la sémantique opérationnelle décrite plus haut.
Le projet est à rendre **avant le** :

15 novembre 2023 à 19h59

Pour finir, vous devez vous assurer des points suivants :

- Le projet contenu dans cette archive **doit compiler**.
- Vous devez **être les auteurs** de ce projet.
- Il doit être rendu **à temps**.

Si l'un de ces points n'est pas respecté, la note de 0 vous sera affectée.

4 Log

25-10-2023 Version initiale.