Projet du cours « Compilation »
Jalon 3 : Typage de HOPIX

version numéro 1.0

1 Système de types

1.1 Syntaxe des types et des environnements de typage

On se donne la syntaxe qui suit pour les types et les environnements de typage de *Hopix*.

$\tau ::=$	**Types**
$\mid \alpha$	Variable de type
$\mid T\ (\tau_1, \ldots, \tau_n)$	Type construit
$\mid \tau_1 \to \tau_2$	Type fonctionnel
$\mid (\tau_1 \star \ldots \star \tau_n)$	Type produit
$\sigma ::=$	**Schémas de type**
$\mid \forall\, \overline{\alpha}.\, \tau$	Quantification
$\Gamma ::=$	**Contextes de typage**
$\mid \bullet$	Contexte vide
$\mid \Gamma, (x : \sigma)$	Déclaration de variable de terme
$\mid \Gamma, \alpha$	Déclaration de variable de type
$\mid \Gamma, (T(\overline{\alpha}) = \Sigma_{i\in[1,n]} K_i : \overline{\tau_i})$	Déclaration de type enregistrement
$\mid \Gamma, (T(\overline{\alpha}) = \{\ell_1 : \tau_1; \ldots; \ell_n : \tau_n\})$	Déclaration de type enregistrement
$\mid \Gamma, (T(\overline{\alpha}) = \mathit{abstract})$	Déclaration de type abstrait

où τ est un type, α est une variable de type, σ est un schéma de type et T est un constructeur de type. Les constructeurs de type comprennent les types prédéclarés comme **int**, **bool** ou **ref** ainsi que les types définis par le programmeur, comme les enregistrements et sommes.

On rappelle que la notation $\overline{\alpha}$ représente une séquence potentiellement vide de α. Pour simplifier les notations, on écrira τ pour un monotype, c'est-à-dire un schéma de type dont les paramètres $\overline{\alpha}$ sont vides.

Dans la syntaxe des environnements de typage, $(x : \sigma)$ signifie que x a le schéma de type σ ; α signifie que la variable de type α est liée ; $(T(\overline{\alpha}) := \Sigma_{i\in[1..n]} K_i : \overline{\tau}_i)$ signifie que le constructeur de type T est un type somme paramétré par les variables de type $\overline{\alpha}$ et que ses constructeurs de données sont les $\overline{K}_i$ de paramètres $\overline{\tau}_i$; et enfin, $(T(\overline{\alpha}) := \{\ell_1 : \tau_1; \ldots; \ell_n : \tau_n\})$ signifie que le constructeur de T est un type enregistrement paramétré par les variables de type $\overline{\alpha}$ et que ses étiquettes ℓ_i ont pour paramètres τ_i.

1.2 Opérations sur les types, les schémas de type et les environnements

Le module `HopixTypes` fournit un ensemble d'opérations utiles. On décrit les principales dans cette section.

Syntaxe interne des types Les types de l'AST de Hopix sont annotés avec les positions du code source, mais ces positions vont nous gêner lors de la vérification des types. Par exemple, déterminer si deux types sont égaux exige d'ignorer les positions.

Pour résoudre ce problème, on introduit une *syntaxe interne* des types (voir le type `HopixTypes.aty`) qui suit la même structure mais ne contient pas les positions.

De nombreuses opérations sont fournies sur ces types : affichage, substitution, calcul des variables de type libres, etc. Par ailleurs, les types prédéfinis du langage (**bool**, **int**, etc.) sont aussi déclarés dans ce module.

Opérations sur les schémas de type Soit un schéma de type de la forme $\forall\, \overline{\alpha}.\, \tau$. On peut *instancier* ce schéma en substituant les variables de type $\overline{\alpha}$ par une liste de (mono)types $\overline{\tau}'$ de même longueur. On dit que le type $\tau[\overline{\alpha}\backslash\overline{\tau'}]$ est une *instance* du schéma de types $\forall\, \overline{\alpha}.\, \tau$. Cette opération est réalisée par la fonction `HopixTypes.instantiate_type_scheme`.

Jugement	Direction	Signification
$\Gamma \vdash_{def} d \Rightarrow \Gamma'$	*Synthèse*	La définition d est bien formée dans le contexte Γ et produit un contexte étendu Γ'.
$\Gamma \vdash_{exp} e \Rightarrow \tau$	*Synthèse*	L'expression e a le type τ dans le contexte Γ.
$\Gamma \vdash_{exp} e \Leftarrow \tau$	*Vérification*	L'expression e a le type τ dans le contexte Γ.
$\Gamma \vdash_{pat} p \Rightarrow \tau \dashv \Gamma'$	*Synthèse*	Le motif p filtre des valeurs de type τ dans le contexte Γ et produit le contexte étendu Γ'.
$\Gamma \vdash_{pat} p \Leftarrow \tau \dashv \Gamma'$	*Vérification*	Le motif p filtre des valeurs de type τ dans le contexte Γ et produit le contexte étendu Γ'.
$\Gamma \mid \tau \vdash_{br} b \Rightarrow \tau'$	*Synthèse*	Si la branche b filtre le type τ dans le contexte Γ il en résulte une valeur de type τ'.
$\Gamma \mid \tau \vdash_{br} b \Leftarrow \tau'$	*Vérification*	Si la branche b filtre le type τ dans le contexte Γ il en résulte une valeur de type τ'.
$\Gamma \vdash_{type} \sigma$	*Vérification*	Le schéma de types σ est bien formé dans le contexte Γ.

FIGURE 1 – Jugements de typage

Construction et déconstruction des types Les opérations de construction et destruction de types permettent de manipuler facilement les types lors du processus de vérification de types. En particulier, les fonctions de destruction sont les inverses partielles des fonctions de construction. Elles permettent d'accéder aux constituants d'un type, en supposant que celui-ci est d'une forme donnée, et lèvent une exception sinon. Par exemple, `destruct_function_type` τ renvoie la paire (τ_1, τ_2) si $\tau = \tau_1 \to \tau_2$, et lève une exception si τ n'est pas un type fonctionnel.

Opérations sur les environnements de typage Le type `typing_environment` est celui des contextes de typage, aussi appelés *environnements* de typage. Il contient toutes les informations nécessaires : déclarations de variables, défintions des types sommes et enregistrements, etc.

Le module fournit de nombreuses opérations sur les environnements de typage : introduction d'une définition de types, d'une variable de type, d'un identificateur de valeurs, recherche d'informations sur ces objets, etc. Le type des environnements étant abstrait, vous devez passer par ces fonctions pour le manipuler.

L'environnement doit obéir à un invariant indispensable durant la phase de typage : les types qui apparaissent dans l'environnement doivent être *bien formés*. La fonction `internalize_ty` transforme un type externe (`HopixAST.ty`) en type interne (`HopixTypes.aty`) en vérifiant sa bonne formation dans le contexte. Assurez-vous de lire et comprendre le code de la fonction interne `check_well_formed_type`.

Environnement de typage initial L'environnement de typage initial est aussi fourni : étudiez sa définition !

1.3 Règles de typage bidirectionnelles

Cette section fournit une partie des règles de typage. Vous devez spécifier les règles manquantes avant de les implémenter.

Le système de type de HOPIX est défini par plusieurs jugements de typage : il y a un jugement pour les définitions, un pour les expressions, et un pour les motifs. De plus, le typage des expressions et des motifs est bidirectionnel, et se décompose en un jugement de *synthèse* et un jugement de *vérification*. La figure 1 résume la forme et la signification des cinq jugements utilisés.

Typage des définitions Les définitions sont toutes traitées suivant leur ordre d'apparition dans le programme, exceptées les définitions de fonctions mutuellement récursives qui doivent être traitées ensemble (voir la prochaine sous-section). Le traitement des valeurs simples peut fonctionner en mode vérification ou en mode synthèse, en fonction de la présence d'une annotation de schéma de types. Ce fonctionnement est exprimé par ces règles.

$$\frac{\Gamma \vdash_{type} \forall\,\overline{\alpha}.\,\tau \qquad \Gamma, \overline{\alpha} \vdash_{exp} e \Leftarrow \tau \qquad \overline{\alpha} \notin FTV(\Gamma)}{\Gamma \vdash_{def} \textbf{val}\ (x : \forall\,\overline{\alpha}.\,\tau) = e \Rightarrow \Gamma, (x : \forall\,\overline{\alpha}.\,\tau)}\ \textsc{CheckSimpleVal}$$

$$\frac{\Gamma \vdash_{exp} e \Rightarrow \tau}{\Gamma \vdash_{def} \textbf{val}\ x = e \Rightarrow \Gamma, (x : Gen_\Gamma(\tau))}\ \textsc{SynthSimpleVal}$$

La première exprime que pour pouvoir ajouter la définition de x dans l'environnement de typage, il faut vérifier que le schéma de type $\forall\,\overline{\alpha}.\,\tau$ écrit par l'utilisateur est bien formé et que l'expression e qui définit x est bien de type τ. Pour avoir le droit de généraliser les variables de type, il faut qu'elles n'apparaissent pas libres dans Γ. En l'absence d'annotation, on synthétise le type de l'expression, qu'on généralise à l'aide de la fonction `HopixTypes.generalize_type`, dont la formulation mathématique est

$$Gen_\Gamma(\tau) = \forall\,\overline{\alpha}.\,\tau \text{ où } \overline{\alpha} = FTV(\tau) \setminus FTV(\Gamma).$$

Le typage d'un bloc de fonctions $\overline{f}$ mutuellement récursives n'est possible que si celles-ci sont annotées avec leurs schémas de types. Pour les typer, on commence par vérifier que les annotations de types écrites par l'utilisateur sont bien formées. Ensuite, on produit tout de suite l'environnement Γ' dans lequel les fonctions $\overline{f}$ sont associées à leurs types, *puis* on vérifie qu'effectivement les définitions de ces fonctions sont bien du type annoté par le programmeur.

$$\textsc{SynthVar}\ \frac{\Gamma \vdash_{type} \overline{\tau} \qquad \Gamma \ni x : \forall\, \alpha.\, \tau'}{\Gamma \vdash_{exp} x^{\overline{\tau}} \Rightarrow \tau'[\overline{\alpha}\backslash\overline{\tau}]} \qquad \textsc{SynthInt}\ \frac{}{\Gamma \vdash_{exp} n \Rightarrow \mathbf{int}} \qquad \textsc{SynthBool}\ \frac{}{\Gamma \vdash_{exp} b \Rightarrow \mathbf{bool}}$$

$$\textsc{SynthApp}\ \frac{\Gamma \vdash_{exp} e_1 \Rightarrow \tau_1 \to \tau_2 \qquad \Gamma \vdash_{exp} e_2 \Leftarrow \tau_1}{\Gamma \vdash_{exp} e_1\ e_2 \Rightarrow \tau_2}$$

$$\textsc{SynthFun}\ \frac{\Gamma \vdash_{pat} p \Rightarrow \tau \dashv \Gamma' \qquad \Gamma' \vdash_{exp} e \Rightarrow \tau'}{\Gamma \vdash_{exp} \backslash p \texttt{ => } e \Rightarrow \tau \to \tau'} \qquad \textsc{SynthLet}\ \frac{\Gamma \vdash_{def} d \Rightarrow \Gamma' \qquad \Gamma' \vdash_{exp} e \Rightarrow \tau}{\Gamma \vdash_{exp} d \texttt{ ; } e \Rightarrow \tau}$$

$$\textsc{SynthRecord}\ \frac{\sigma \in \mathfrak{S}_n \qquad \Gamma \vdash_{type} \overline{\tau} \qquad \Gamma \ni T(\overline{\alpha}) = \{\ell_{\sigma(1)} : \tau'_{\sigma(1)}; \ldots; \ell_{\sigma(n)} : \tau'_{\sigma(n)}\} \qquad (\Gamma \vdash_{exp} e_i \Leftarrow \tau'_{\sigma(i)}[\overline{\alpha}\backslash\overline{\tau}])_{i\in[1,n]}}{\Gamma \vdash_{exp} \{\ell_1 \texttt{ = } e_1 \texttt{ ; } \cdots \texttt{ ; } \ell_n \texttt{ = } e_n\}^{\overline{\tau}} \Rightarrow T(\overline{\tau})}$$

$$\textsc{SynthAnnot}\ \frac{\Gamma \vdash_{type} \tau \qquad \Gamma \vdash_{exp} e \Leftarrow \tau}{\Gamma \vdash_{exp} (e : \tau) \Rightarrow \tau} \qquad \textsc{CheckSynth}\ \frac{\Gamma \vdash_{exp} e \Rightarrow \tau}{\Gamma \vdash_{exp} e \Leftarrow \tau}$$

$$\textsc{CheckFun}\ \frac{\Gamma \vdash_{pat} p \Leftarrow \tau \dashv \Gamma' \qquad \Gamma' \vdash_{exp} e \Leftarrow \tau'}{\Gamma \vdash_{exp} \backslash p \texttt{ => } e \Leftarrow \tau \to \tau'} \qquad \textsc{CheckLet}\ \frac{\Gamma \vdash_{def} d \Rightarrow \Gamma' \qquad \Gamma' \vdash_{exp} e \Leftarrow \tau}{\Gamma \vdash_{exp} d \texttt{ ; } e \Leftarrow \tau}$$

$$\textsc{CheckMatch}\ \frac{\Gamma \vdash_{exp} e \Rightarrow \tau' \qquad (\Gamma \mid \tau' \vdash_{br} b_i \Leftarrow \tau)_{i\in[1,n]}}{\Gamma \vdash_{exp} \mathbf{match}\ (\ e\ )\ \{\ b_1, \ldots, b_n\ \} \Leftarrow \tau}$$

FIGURE 2 – Typage des expressions (extrait)

$$\textsc{SynthPInt}\ \frac{}{\Gamma \vdash_{pat} n \Rightarrow \mathbf{int} \dashv \Gamma} \qquad \textsc{SynthPBool}\ \frac{}{\Gamma \vdash_{pat} b \Rightarrow \mathbf{bool} \dashv \Gamma}$$

$$\textsc{SynthPTuple}\ \frac{(\Gamma_{i-1} \vdash_{pat} p_i \Rightarrow \tau_i \dashv \Gamma_i)_{i\in[1,n]}}{\Gamma_0 \vdash_{pat} (\ p_1, \ldots, p_n\ ) \Rightarrow (\tau_1 \star \ldots \star \tau_n) \dashv \Gamma_n} \qquad \textsc{SynthPAnnot}\ \frac{\Gamma \vdash_{type} \tau \qquad \Gamma \vdash_{pat} p \Leftarrow \tau \dashv \Gamma'}{\Gamma \vdash_{pat} (p : \tau) \Rightarrow \tau \dashv \Gamma'}$$

$$\textsc{CheckSynthPat}\ \frac{\Gamma \vdash_{pat} p \Rightarrow \tau \dashv \Gamma'}{\Gamma \vdash_{pat} p \Leftarrow \tau \dashv \Gamma'} \qquad \textsc{CheckPWildcard}\ \frac{}{\Gamma \vdash_{pat} \_ \Leftarrow \tau \dashv \Gamma} \qquad \textsc{SynthPVar}\ \frac{}{\Gamma \vdash_{pat} x \Leftarrow \tau \dashv \Gamma, (x : \tau)}$$

$$\textsc{CheckPTuple}\ \frac{(\Gamma_{i-1} \vdash_{pat} p_i \Leftarrow \tau_i \dashv \Gamma_i)_{i\in[1,n]}}{\Gamma_0 \vdash_{pat} (\ p_1, \ldots, p_n\ ) \Leftarrow (\tau_1 \star \ldots \star \tau_n) \dashv \Gamma_n}$$

FIGURE 3 – Typage des motifs (extrait)

Typage des expressions Le typage des expressions peut être réalisé en mode synthèse ou en mode vérification. Une partie des règle est donnée à la figure 2.

Plusieurs constructions requièrent la présence d'annotations permettant d'instancier un schéma de types. C'est notamment le cas des variables (SYNTHVAR) ou des types construits (enregistrements, types sommes).

La règle SYNTHRECORD exhibe une difficulté spécifique. La création d'une valeur enregistrement n'a pas à lister les champs dans le même ordre que lors de la définition du type, et il faut donc raisonner à permutation près. La notation $\mathfrak{S}_n$ désigne l'ensemble des permutations de l'ensemble $\{1, \ldots, n\}$. La troisième prémisse de la règle indique que le i-ème champ de la création de valeur correspond au $\sigma(i)$-ème champ de sa déclaration. L'expression e_i doit donc être du type de ce champ, c'est-à-dire $\tau'_{\sigma(i)}$, auquel a été appliqué la substitution $\overline{\alpha}\backslash\overline{\tau}$.

Typage des motifs Un motif bien typé *filtre* les valeur d'un certain type et, ce faisant, étend un environnement Γ en un environnement Γ'. Une partie des règle est donnée à la figure 3.

2 Implémentation d'un vérificateur de type

Votre vérificateur va s'appuyer sur des annotations de type écrites par le programmeur. En fonction de votre algorithme, certaines annotations vont être obligatoires, d'autres non.

Dans nos tests, nous allons supposer que :

— toutes les définitions mutuellement récursives sont totalement annotées ;
— toutes les fonctions anonymes sont annotées par leurs types ;
— toutes les expressions qui manipulent les types construits sont annotées avec les types arguments de ceux-ci (cf. les règles SYNTHVAR ou SYNTHRECORD, par exemple) ;

mais il se peut que votre vérificateur demande au programmeur d'écrire moins de types.

Vous devez compléter la fonction `HopixTypechecker.typecheck` qui attend un environnement de typage, une liste de définitions (i.e. un programme) et qui produit l'environnement de typage qui correspond à l'introduction de ces définitions dans l'environnement, à la condition que ces dernières soient bien typées. Si le programme est mal typé alors vous devez utiliser la fonction `HopixTypechecker.type_error` pour produire un message d'erreur. Notez que beaucoup des erreurs sont en fait produites par les fonctions du module `HopixTypes`, que vous devez lire avec attention.

Nous vous suggérons fortement d'esquisser une formulation mathématiques des règles qui manquent aux figures 2 and 3 avant de vous lancer dans leur implémentation. Attention, certaines constructions linguistiques peuvent être traitées en mode synthèse et en en mode vérification, tandis que d'autres ne fonctionnent qu'en mode synthèse ou en mode vérification.

3 Travail à effectuer

La troisième partie du projet est la réalisation d'un vérificateur de types bidirectionnel pour Hopix.
Le projet est à rendre **avant le** :

6 décembre 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

15-11-2023 Version initiale.