# Travail pratique #3

IFT-2035

June 22, 2020

¡¡ Dû le 12 juillet à 23h59 !!

## 1 Survol

Ce TP a pour but de vous familiariser avec le langage Prolog, ainsi qu'avec les macros.

Comme pour le TP précédent, les étapes sont les suivantes:

1. Parfaire sa connaissance de Prolog.
2. Lire et comprendre cette donnée.
3. Lire, et comprendre le code fourni.
4. Compléter le code fourni.
5. Écrire un rapport. Il doit décrire **votre** expérience pendant les points précédents: problèmes rencontrés, surprises, choix que vous avez dû faire, options que vous avez sciemment rejetées, etc... Le rapport ne doit pas excéder 5 pages.

Ce travail est à faire en groupes de 2 étudiants. Le rapport, au format LaTeX exclusivement (compilable sur `frontal.iro`) et le code sont à remettre par remise électronique avant la date indiquée. Aucun retard ne sera accepté. Indiquez clairement votre nom au début de chaque fichier.

Si un étudiant préfère travailler seul, libre à lui, mais l'évaluation de son travail n'en tiendra pas compte. Des groupes de 3 ou plus sont **exclus**.

## 2 Le langage SF

Vous allez écrire un programme Prolog qui va manipuler des expressions du langage du futur SF. C'est un langage fontionnel typé statiquement, avec polymorphisme paramétrique, et avec macros.

Votre travail sera d'implanter le vérificateur de types ainsi que la phase d'élaboration qui fait l'expansion des macros et qui ajoute un peu de sucre syntaxique.

Les types du langage ont la syntaxe suivante:

$$\tau ::= ct \mid \mathsf{list}(\tau) \mid \tau_1 \to \tau_2 \mid \alpha \mid \mathsf{forall}(\alpha, \tau)$$

où *ct* représente n'importe lequel des quelques types atomiques prédéfinis, qui sont: int pour les entiers, bool pour les booléens, macroexpander pour les macros, sexp qui est le type utilisé pour les expressions du code source manipulées par les macros. forall est le type donnés aux expressions polymorphes, comme discuté ci-dessous.

Les expressions du langage ont la syntaxe suivante:

$$e ::= c \mid x \mid \mathsf{fn}(x, e) \mid \mathsf{app}(e_1, e_2) \mid \mathsf{quote}(e) \mid \mathsf{if}(e_1, e_2, e_3) \mid \mathsf{tfn}(\alpha, e) \mid \mathsf{tapp}(\tau, e)$$

où *c* représente n'importe quelle constante prédéfinie, cela inclus par exemple les nombres entiers et les opérations sur les listes; fn et app définissent et appellent les fonctions; quote permet d'inclure un morceau de code comme une constante (de type sexp).

tfn et tapp définissent les expressions polymorphes, qui fonctionnent comme des fonctions sauf qu'elles prennent des types comme arguments. Par exemple, la fonction identité polymorphe *identity* s'écrit $\mathsf{tfn}(t, \mathsf{fn}(x, x))$, elle a type $\mathsf{forall}(t, t \to t)$, et pour l'appliquer il faut d'abord la spécialiser avec tapp, e.g.: $\mathsf{app}(\mathsf{tapp}(\mathit{identity}, \mathsf{int}), 42)$. Autant tfn que tapp ne servent qu'au typage et n'ont pas d'effet à l'exécution, où $\mathsf{tfn}(\alpha, e)$ et $\mathsf{tapp}(e, \tau)$ se réduisent simplement à $e$.

Un programme SF est constitué d'une séquence de déclarations suivie par une expression principale à évaluer. Les déclarations ont la forme suivante:

$$\mathit{decl} ::= x : \tau \mid \mathsf{define}(x, e)$$

où la forme $x : \tau$ sert d'une part d'annotation de type, comme en Haskell, mais aussi de déclaration avancée pour permettre de vérifier le type des définitions récursives.

La syntaxe du langage repose sur la syntaxe de Prolog, donc par exemple, $\tau_1 \to \tau_2$ peut aussi s'écrire $\to (\tau_1, \tau_2)$, et de même $x : \tau$ n'est rien d'autre qu'une écriture alternative de $: (x, \tau)$.

Le langage inclut des opérations sur les listes à la Lisp, avec des primitives comme *cons*, *car*, *cdr*, *nil*. À l'exécution, ces listes sont représentées par des listes Prolog.

# 3 Inférence de types

SF utilise un typage statique similaire à celui de Haskell, où les types sont largement inférés. Il est un peu moins sophistiqué que le système utilisé en Haskell.

Les règles n'incluent pas les opérations sur les listes, car le polymorphisme de SF permet de les traiter simplement comme des constantes prédéfinies dans l'environnement initial.

$$\frac{}{\Gamma \vdash n \ : \ \mathsf{int}} \qquad \frac{\Gamma(x) = \tau}{\Gamma \vdash x \ : \ \tau} \qquad \frac{\Gamma \vdash e \ : \ \mathsf{bool} \quad \Gamma \vdash e_t \ : \ \tau \quad \Gamma \vdash e_e \ : \ \tau}{\Gamma \vdash \mathsf{if}(e, e_t, e_e) \ : \ \tau}$$

$$\frac{\Gamma, x{:}\tau_1 \vdash e \ : \ \tau_2}{\Gamma \vdash \mathsf{fn}(x, e) \ : \ \tau_1 \to \tau_2} \qquad \frac{\Gamma \vdash e_1 \ : \ \tau_1 \to \tau_2 \quad \Gamma \vdash e_2 \ : \ \tau_1}{\Gamma \vdash \mathsf{app}(e_1, e_2) \ : \ \tau_2}$$

$$\frac{\Gamma, \alpha{:}\mathsf{type} \vdash e \ : \ \tau}{\Gamma \vdash \mathsf{tfn}(\alpha, e) \ : \ \mathsf{forall}(\alpha, \tau)} \qquad \frac{\Gamma \vdash e \ : \ \mathsf{forall}(\alpha, \tau_2) \quad \Gamma \vdash \tau \ : \ \mathsf{type}}{\Gamma \vdash \mathsf{tapp}(e, \tau_1) \ : \ \tau_2[\tau_1/\alpha]}$$

$$\frac{}{\Gamma \vdash \mathsf{quote}(e) \ : \ \mathsf{sexp}} \qquad \frac{\Gamma \vdash \tau \ : \ \mathsf{type}}{\Gamma \vdash \mathsf{list}(\tau) \ : \ \mathsf{type}}$$

$$\frac{\Gamma \vdash \tau_1 \ : \ \mathsf{type} \quad \Gamma \vdash \tau_2 \ : \ \mathsf{type}}{\Gamma \vdash \tau_1 \to \tau_2 \ : \ \mathsf{type}} \qquad \frac{\Gamma, \alpha{:}\mathsf{type} \vdash \tau \ : \ \mathsf{type}}{\Gamma \vdash \mathsf{forall}(\alpha, \tau) \ : \ \mathsf{type}}$$

Figure 1: Règles de typage

# 4 Élaboration

Pour rendre l'écriture de programmes plus commode, le code source passe par une phase d'*élaboration* avant d'être envoyé à l'inférence de types.

L'élaboration inclus 3 éléments: l'expansion des appels de macro, l'expansion des appels de fonction en forme compacte, et finalement l'introduction des paramètres de types.

## 4.1 Paramètres de types

En pratique, les paramètres de type passés aux fonctions comme *identity* sont non seulement incommodes, mais en grande partie redondants. Comme le montre Haskell, on peut presque toujours les inférer. En fait Haskell infère non seulement les `tapp` mais aussi les `tfn`. SF lui se contente d'inférer les `tapp`. Plus précisément, la phase d'élaboration va remplacer chaque usage d'une fonction polymorphe par une expression qui spécialise cette fonction à un type non-spécifié qui sera trouvé par l'inférence de types. E.g. chaque usage de *cons* sera remplacé automatiquement par `tapp`(*cons*, ?), où le ? représente le fait que cette partie du code sera remplie plus tard par l'inférence de types.

## 4.2 Appels de fonction en forme compacte

Pour rende la syntaxe un peu moins lourde, l'élaboration va considérer toute expression de la forme $\mathit{fun}(e_1, e_2, ..., e_n)$ (sauf celles qui ont déjà une signification particulière) comme du sucre syntaxique pour $\mathsf{app}(..\mathsf{app}(\mathsf{app}(\mathit{fun}, e_1), e_2)..., e_n)$.

Donc, par exemple une expression $cons(1, nil)$ sera transformée par élaboration en $\mathsf{app}(\mathsf{app}(\mathsf{tapp}(cons, ?), 1), \mathsf{tapp}(nil, ?))$, suite à quoi l'inférence de type la changera en $\mathsf{app}(\mathsf{app}(\mathsf{tapp}(cons, \mathsf{int}), 1), \mathsf{tapp}(nil, \mathsf{int}))$.

### 4.3 Macros

SF offre un système de macros similaire à Lisp. Une macro est une variable dont la valeur est un objet de type macroexpander, construit avec le constructeur macro qui prend un argument qui est une fonction de type $\mathsf{list}(\mathsf{sexp}) \rightarrow \mathsf{sexp}$. Toute expression de la forme $mac(e_1, e_2, ..., e_n)$, où $mac$ est une macro définie dans l'environnement, est considérée comme un appel de macro et sera remplacé par son expansion qui est le résultat de l'appel à la fonction qui défini la macro, avec comme argument la liste $[e_1, e_2, ..., e_n]$ (il s'agit bien ici d'une liste d'expressions non évaluées du code source, donc un objet de type $\mathsf{list}(\mathsf{sexp})$).

## 5 Votre travail

Votre travail consiste en deux parties: d'une part l'implémentation de l'inférence de type, et d'autre part l'implémentation de la phase d'élaboration.

Le code qui vous est fourni contient déjà l'évaluateur ainsi que la boucle principale qui lit les différentes déclarations du programme SF. Il contient aussi une règle de substitution que vous pouvez utiliser pour $\tau_2[\tau_1/x]$.

## 6 Remise

Vous devez remettre deux fichiers, `sf.pl` et `rapport.tex`, qui seront remis sous forme électronique, par l'intermédiare de Moodle/StudiUM.

## 7 Détails

La note est basée d'une part sur des tests automatiques, d'autre part sur la lecture du code, ainsi que sur le rapport. Le critère le plus important, et que votre code doit se comporter de manière correcte. En règle générale, une solution simple et plus souvent correcte qu'une solution complexe. Ensuite, vient la qualité du code: plus c'est simple, mieux c'est. S'il y a beaucoup de commentaires, c'est généralement un symptôme que le code n'est pas clair; mais bien sûr, sans commentaires le code (même simple) et souvent incompréhensible. L'efficacité de votre code est sans importance, sauf s'il utilise un algorithme vraiment particulièrement ridiculement inefficace.

Les points seront répartis comme suit: 25% sur le rapport, 25% sur les tests automatiques, 25% sur le code de l'inférence de type, et 25% sur le code de l'élaboration.

- Le code ne doit en aucun cas dépasser 80 colonnes.

- Vérifiez la page web du cours, pour d'éventuels errata, et d'autres indications supplémentaires.
- Tout usage de matériel (code ou texte) emprunté à quelqu'un d'autre (trouvé sur le web, ...) doit être dûment mentionné, sans quoi cela sera considéré comme du plagiat.