Travail pratique #2

IFT-2035

6 juin 2023

1 Survol

Ce TP vise à améliorer la compréhension des langages fonctionnels et de la métaprogrammation, en utilisant un langage de programmation fonctionnel (Haskell) et en écrivant une partie d'un interpréteur d'un langage de programmation fonctionnel (en l'occurrence une sorte de Lisp). Les étapes de ce travail sont les suivantes :

1. Parfaire sa connaissance de Haskell et des macros.
2. Lire et comprendre cette donnée.
3. Lire, trouver, et comprendre les parties importantes du 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 `ens.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**.

τ ::=	Int	Type des nombres entiers
	| String	Type des chaînes de caractères
	| Bool	Type des booléens
	| Sexp	Type des arbres de syntaxe
	| Macro	Type des macros
	| SpecialForm	Type des *formes spéciales*
	| $(\tau_1 \ ... \ \tau_n \rightarrow \tau)$	Type d'une fonction
e ::=	n	Un entier signé en décimal
	| x	Référence à une variable
	| $(e_0 \ e_1 \ ... \ e_n)$	Un appel de fonction (*curried*)
	| (fun x e)	Une fonction d'un argument
	| (let $((x_1 \ e_1)...(x_n \ e_n))$ e_b)	Ajout de variables locales
	| (: e τ)	Annotation de type
	| $+$ | $-$ | $*$ | $/$	Opérations arithmétiques
	| (if e_c e_v e_f)	Expression conditionnelle
	| *zero?* | *true* | *false* | ...	Booléens
	| $(m \ e_1 \ ... \ e_n)$	Appel de macro
	| $'e$	Valeur Sexp litérale
	| *nil* | *cons* | *car* | *cdr* | ...	Manipulation de Sexp
d ::=	(dec x τ)	Déclaration de variable
	| (def x e)	Définition de variable
	| $(m \ e_1 \ ... \ e_n)$	Appel de macro
p ::=	$d_1 \ ... \ d_n$	Programme

FIGURE 1 – Syntaxe de Psil

2 Psil : Une sorte de Lisp

Vous allez travailler sur l'implantation d'une version étendue de Psil. La syntaxe de ce langage est décrite à la Figure 1.

Par rapport au TP1, les changements sont les suivants :

— La fonctions *if0* n'existe plus. À la place, le langage a acquis les booléens et la fonction de test *zero?*.

— La forme let accepte maintenant un nombre arbitraire de variables locales, qui sont traitées en séquence, i.e. la définition de chaque variable peut faire référence aux variables qui la précédent.

— Le langage offre maintenant des macros similaires à celles de ELisp. Pour définir la variable m comme une macro, il suffit de lui donner une valeur de type Macro. On construit une telle valeur avec le constructeur macro qui est une fonction prédéfinie de type ((Sexp → Sexp) → Macro). Ces macros n'acceptent qu'un argument et si elles veulent en prendre plus, elles doivent alors renvoyer une Sexp spéciale construite avec le constructeur moremacro de type ((Sexp → Sexp) → Sexp). Tout comme en ELisp,

les macros manipulent des arbres de syntaxe de type Sexp.

— Il n'y a plus de mots clé dans la syntaxe des expressions : e.g. fun n'est plus un mot réservé. À la place, *fun* est une variable prédéfinie dont la valeur est une *forme spéciale*. Alors qu'une macro transforme une Sexp en une autre Sexp, une forme spéciale implémente une construction fondamentale du langage qui ne peut pas s'exprimer comme une macro. La différence avec l'usage de mots clé et que les formes spéciales ne sont pas irrémédiablement attachées à leur nom, donc on peut leur donner un autre nom et utiliser le nom original pour autre chose. E.g. on peut "franciser" le langage avec :

```
(def si if)
(def soit let)
(def fon fun)
```

2.1 Sucre syntaxique

La syntaxe des arbres du code source est un peu plus rafinée qu'au TP1 :

$$\begin{aligned}
(v_1\ .\ v_2\ ...\ v_n) &\iff ((v_1\ .\ v_2)\ .\ ...\ v_n)\\
(v_1\ ...\ v_n) &\iff (()\ .\ v_1\ ...\ v_n)\\
'v &\iff (\text{shorthand-quote}\ v)\\
`v &\iff (\text{shorthand-backquote}\ v)\\
,v &\iff (\text{shorthand-comma}\ v)
\end{aligned}$$

Cela se marie très naturellement avec le *currying* des appels de fonctions qui en dérive presque inévitablement. On retrouve donc le même comportement que pour le TP1, sauf que l'équivalence suivante s'applique plus généralement :

$$(e_1\ e_2) \iff (e_1\ .\ e_2)$$

Par conséquent, on obtient des équivalences moins communes (et à vrai dire, probablement moins intuitives) comme :

$$\begin{aligned}
(\text{let}\ d\ e_1\ e_2) &\iff ((\text{let}\ d\ e_1)\ e_2)\\
(\text{if}\ c\ e_1\ e_2\ e_3) &\iff ((\text{if}\ c\ e_1\ e_2)\ e_3)\\
(\text{fun}\ x\ e_1\ e_2) &\iff ((\text{fun}\ x\ e_1)\ e_2)
\end{aligned}$$

Plus important, les appels de macro sont aussi *curried* :

$$(m\ e_1\ e_2) \iff ((m\ e_1)\ e_2)$$

Donc les macros ne prennent fondamentalement qu'un argument. Pour définir des macros de plusieurs arguments, il faut que la macro renvoie une expansion spéciale de la forme *moremacro* à laquelle sera passé l'argument suivant. E.g. :

```
(def macro-a-3-arguments
     (macro
      (fun arg1
```

sinthétiser vérifier

$$\frac{}{\Gamma \vdash n \Rightarrow \mathsf{Int}} \qquad \frac{\Gamma(x) = \tau}{\Gamma \vdash x \Rightarrow \tau} \qquad \frac{\Gamma \vdash e \Leftarrow \tau}{\Gamma \vdash (:\, e\ \tau) \Rightarrow \tau} \qquad \frac{\Gamma \vdash e \Rightarrow \tau}{\Gamma \vdash e \Leftarrow \tau}$$

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

$$\frac{\Gamma \vdash e_b \Leftrightarrow \tau_b}{\Gamma \vdash (\mathsf{let}\ ()\ e_b) \Leftrightarrow \tau_b} \qquad \frac{\Gamma \vdash e_1 \Rightarrow \tau_1 \quad \Gamma, x_1{:}\tau_1 \vdash (\mathsf{let}\ (d)\ e_b) \Leftrightarrow \tau_b}{\Gamma \vdash (\mathsf{let}\ ((x_1\ e_1)\ d)\ e_b) \Leftrightarrow \tau_b}$$

$$\frac{\Gamma \vdash e_c \Leftarrow \mathsf{Bool} \quad \Gamma \vdash e_v \Leftarrow \tau \quad \Gamma \vdash e_f \Leftarrow \tau}{\Gamma \vdash (\mathsf{if}\ e_c\ e_v\ e_f) \Leftarrow \tau} \qquad \frac{}{\Gamma \vdash {'e} \Rightarrow \mathsf{Sexp}}$$

FIGURE 2 – Règles de typage

```
(moremacro
 (fun arg2
      (moremacro
       (fun arg3 <BODY>)))))))
```

où `<BODY>` calculera le résultat de l'expansion des appels de macro de la forme (`macro-a-3-arguments` e_1 e_2 e_3). Si vous trouvez ça malpratique, n'oubliez que le fait que le mécanisme est primitif permet une implantation plus simple, donc vous en bénéficierez. De plus on peut cacher cette lourdeur derrière une couche de macros.

2.2 Sémantique statique

Le typage reste presque le même que pour le TP1, mis à part l'ajout de if et l'extension de let. La figure 2 rappelle ces règles, qui sont encore bidirectionnelles.

Un changement supplémentaire est que let peut-être utilisé autant dans le sens de la vérification que de la synthétisation du type, ce qui est représenté par l'usage de $\Leftrightarrow$, où il est sous entendu que la direction de la flèche doit être la même pour l'hypothèse que pour la conclusion.

2.3 L'environnement d'exécution

Vous avez reçu une nouvelle version du code Haskell, ainsi qu'un fichier de tests Psil sensiblement étendu.

Voilà ci-dessous un exemple de session interactive sur une machine GNU/Linux, avec le code fourni :

```
% ghci psil2.hs
GHCi, version 8.8.4: https://www.haskell.org/ghc/  :? for help
[1 of 1] Compiling Main             ( psil2.hs, interpreted )
```

```
psil2.hs:521:1: warning: [-Wincomplete-patterns]
    Pattern match(es) are non-exhaustive
    In an equation for ‘eval’:
        Patterns not matched:
            _ (Lif _ _ _)
            _ (Lquote _)
    |
521 | eval _venv (Lnum n) = Vnum n
    | ^^^^^^^^^^^^^^^^^^^^^^^^^^^...
Ok, one module loaded.
*Main> run "exemples2.psil"
  2 : Tprim "Int"
  <Fonction> : Tarw (Tprim "Int") (Tarw (Tprim "Int") (Tprim "Int"))
  <Fonction> : Tarw (Tprim "Int") (Tprim "Int")
  <Fonction> : Tarw (Tprim "Int") (Tprim "Int")
  6 : Tprim "Int"
  7 : Tprim "Int"
  *** Exception: ¡¡COMPLÉTER!! sf_let
CallStack (from HasCallStack):
  error, called at psil2.hs:303:23 in main:Main
*Main>
```

Lorsque votre travail sera fini, il ne devrait plus y avoir d'avertissements, et `run` devrait renvoyer plus de valeurs que juste les deux ci-dessus et terminer sans erreurs.

3 À faire

Vous allez devoir compléter diverses parties de `psil2.hs` ainsi qu'une petite partie de `exemples2.psil`. Je recommande de le faire en avançant dans `exemples2.psil`. Ceci dit, libre à vous de choisir l'ordre qui vous plaît.

De même je vous recommande fortement de travailler en binôme (*pair programming*) plutôt que de vous diviser le travail, vu que la difficulté est plus dans la compréhension que dans la quantité de travail.

Le code contient des indications des endroits que vous devez modifiez. Généralement cela signifie qu'il ne devrait pas être nécessaire de faire d'autres modifications, sauf ajouter des définitions auxiliaires. Vous pouvez aussi modifier le reste du code, si vous le voulez, mais il faudra alors justifier ces modifications dans votre rapport en expliquant pourquoi cela vous a semblé nécessaire.

Vous devez aussi fournir un fichier de tests `tests2.psil`, qui, tout comme `exemples2.psil` devrait non seulement contenir du code Psil mais aussi indiquer les valeurs et types qui lui corresponde (à votre avis). Il doit contenir au moins 5 tests que *vous* avez écrits et qui devraient tester différents recoins de Psil.

3.1 Remise

Pour la remise, vous devez remettre trois fichiers (`psil2.hs`, `tests2.psil`, et `rapport2.tex`) par la page Moodle (aussi nommé StudiUM) du cours. Assurez-vous que le rapport compile correctement sur `ens.iro` (auquel vous pouvez vous connecter par SSH).

4 Détails

— La note sera divisée comme suit : 25% pour le rapport, 20% pour le code Psil, 55% pour le code Haskell (i.e. élaboration, vérification des types, et évaluation).
— 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.
— 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.
— 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, est que votre code doit se comporter de manière correcte. 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) est souvent incompréhensible. L'efficacité de votre code est sans importance, sauf si votre code utilise un algorithme vraiment particulièrement ridiculement inefficace.