TP1 - Rapport

Philippe Gabriel
Dana Seif Eddine

30 mai 2021

1 Syntaxe de Psil

La syntaxe de ce langage fonctionnel était assez simple à saisir, surtout après avoir regardé les exemples simples dans le fichier `sample.psil`. Pour la plupart des règles syntaxiques, nous avions presque immédiatement pu associer une sémantique à chacune de celles-ci. La syntaxe étant préfixe a grandement simplifié l'analyse d'expressions et leur conversion.

1.1 Implantation - Conversion de Sexp à Lexp

Un élément important que nous avions fait avant de commencer notre implémentation de code était de rapidement survoler la section du code traitant la première phase déjà fournit, visant à transformer le code source en une représentation `Sexp`. Le but était de se familiariser avec les définitions et fonctions utiles pour compléter la seconde phase. Un premier défi par la suite était de bien comprendre la représentation `Sexp` provenant d'une expression `Psil`. Pour le surmonter, nous avions utilisé la fonction `sexpOf` sur certains exemples de ceux donnés, pour ensuite en refaire la trace et comprendre l'ordre de construction de l'expression en `Psil` vers une `Sexp`.

Par après, la conversion d'une `Sexp` en `Lexp` s'est déroulée assez simplement et rapidement. Après s'être familiariser avec le comportement de la fonction `sexp2list`, la conversion des expressions `if`, `tuple` et `fetch` étaient assez directes.
Pour l'expression `call`, afin de satisfaire son caractère curried, une première solution était de repasser en argument à `s2l` le premier argument du constructeur `Scons`. Pour cela, nous avions modifié le code comme-ci:

```
s2l (se@(Scons se1 _)) = -- Modification ici
    case sexp2list se
        ...
        (Ssym "call" : e : e1 : []) -> Lcall (s2l e) (s2l e1)
        (Ssym "call" : en) -> Lcall (s2l se1) (s2l (last en))
```

Cette implémentation fonctionnait comme souhaité, mais une approche similaire ne fut pas possible pour `fun`. De plus, nous avons réalisé que le temps de calcul pour gérer le cas d'un appel de fonction est loin d'être optimal en raison du fait que l'on repasse une portion de notre `Sexp` à `s2l` pour la reconvertir avec `sexp2list` en une liste, ce qui est du travail inutile puisque l'on possède de la représentation en liste désirée. Il fut donc nécessaire de légèrement remodifier la définition de la fonction principale `s2l` comme suit:

```
s2l (se@(Scons _ _)) =
    let
        selist = sexp2list se -- Modification ici
    in
        ...
        (Ssym "call" : _) -> s2l' se selist
        (Ssym "fun" : _) -> s2l' se selist
```

où nous avions simplement dénoté l'expansion de la `Sexp` sous le nom de la variable `selist`. La raison du changement est pour ensuite faire appel à la fonction `s2l'`, une fonction auxiliaire capable de traiter les cas récursifs que l'on retrouve dans `call` et `fun`.
Finalement, pour l'expression `let`, il a été important de penser aux différentes syntaxes de déclarations qu'offrent le langage `Psil`. Il a donc été nécessaire de définir une fonction `s2d` qui traite les différents cas possibles.

En ce qui concerne les types, les cas de `Bool` et `Tuple` furent assez direct. Par contre, pour le type des fonctions, il fut nécessaire, tout comme dans le cas de `s2l`, de redéfinir la fonction `s2t` en dénotant l'expansion de la `Sexp` en liste et en définissant une fonction auxiliaire `s2t'` pour gérer l'aspect curried du type des fonctions.

1.2 Sucre syntaxique

Après l'implémentation assez direct de la syntaxe du langage, nous nous sommes tournés vers le sucre syntaxique et les différentes équivalences de syntaxe qu'offre le langage.

Pour ce qui est de la syntaxe des expressions, notre définition initiale supportait déjà les différentes variantes acceptées.

Pour ce qui est de la syntaxe des déclarations, au début de ce projet il ne semblait pas nécessaire qu'après être rendu à la fin du projet de modifier notre implantation initiale. Nous avions pris note des différentes équivalences entre déclarations mais ne savions pas tout à fait pourquoi et comment, à ce stade, modifier notre définition qui ne faisait que directement traduire une déclaration en ses `Lexp` et `Ltype` correspondants.
Lors de l'exécution de tests, il y eut des problèmes dans la vérification et l'évaluation de certaines expressions. Nous nous

sommes souvenus de notre note initiale, et avions par après modifier les traductions de déclarations de sorte à les convertir à la forme suivante:

$$
\begin{aligned}
&(x\ (x_1\ \tau_1)\ ...\ (x_n\ \tau_n)\ \tau\ e) && \Longrightarrow \\
&(x\ (\tau_1\ ...\ \tau_n \rightarrow \tau)\ (\text{fun}\ (x_1\ ...\ x_n)\ e)) && \Longrightarrow \\
&(x\ \tau\ e) && \Longrightarrow\ (x\ (\text{hastype}\ e\ \tau))
\end{aligned}
$$

Ce que les implications ci-dessus décrivent:

- C'est que la déclaration d'une fonction peut être réécrite en spécifiant le type sous une forme currifiée et le corps comme étant une fonction;
- Qui à son tour peut être réécrite comme toute déclaration de variable avec son type;
- Qui peut être réécrite l'aide de l'expression `hastype`.

Cette dernière réécriture est celle qui est préférée pour la vérification de types qui suit et donc `s2t` ainsi que `s2t'` ont été redéfinies afin de réécrire la `Sexp` en `Lexp` sous la forme désirée.

2 Règles de typage

Les règles de typages fournies étaient assez intuitives dans le choix de notation, et ont servis de base de la complétion des fonctions `infer` et `check`.

2.1 Implantation - Vérificateur de types

Cette partie de l'implantation s'est déroulée très rapidement. En suivant les règles de typage du langage, il y a cette correspondance directe entre les règles et l'implantation. Au fur et à mesure que l'on introduisait une nouvelle expression, des tests rapides à l'aide la fonction `typeOf` venait confirmer que notre implantation s'est bien faite.

3 Règles d'évaluation

Les règles d'évaluation étaient assez intuitives et les β-réductions appliquées étaient prises en notes lors de cette phase. La seule règle qui ne figurait pas dans les instructions était celle propre aux fonctions (pas l'appel de fonction). Après le survol du code, nous nous sommes basés sur la forme des fonctions prédéfinies du langage pour déterminer la forme attendue de l'évaluation d'une fonction. On note également une correspondance directe des constructeurs de types de `Value` avec ceux de `Ltype` ce qui nous a indiqué précisément le type de résultat de l'évaluation des diverses expressions.

3.1 Implantation - Évaluateur

À la lumière des remarques et notes ci-dessus, l'évaluation des expressions `call`, `let`, `if`, `tuple` et `fetch` fut assez directe et simple. Le cas qui a prouvé être difficile est celui de `fun` pour lequel il n'était pas clair à vue d'oeil comment décrire son évaluation. Comme première idée, nous avions remarqué que le type semble être nécessaire pour l'évaluation d'une fonction, et donc avions modifié le code comme suit:

```
eval2 senv (Lhastype e t) =
    case t of
    Larw _ _ -> eval2fun senv e t
    _ -> eval2 senv e
```

Cette idée semblait être la seule solution possible à ce temps, mais on était pas arrivé à déterminer une forme générale pour les diverses formes de fonctions. C'est alors que nous avions pensé à une alternative, sans avoir besoin du type de la fonction, où la valeur de celle-ci sera déterminée au moment après qu'elle soit définie. Et donc, le code modifié précédemment a été remis tel qu'il était et notre idée alternative sur l'évaluation d'une fonction est comme suit:

```
eval2 senv (Lfun a e) =
    \venv -> (Vfun Nothing (\v -> (eval2 (a : senv) e) (v : venv)))
```

4 Tests

L'étape qui suit à ce point est de tester notre implémentation sur des exemples. Nous avions employé pour commencer les tests qui figurent dans le fichier `sample.psil`. On voyait que tous les tests passaient sauf deux en particulier:

- La vérification de types de l'expression contenant une déclaration de fonction faisant référence à deux déclarations qui la précédait (l'avant- avant dernier exemple);
- La vérification et l'évaluation de l'expression contenant une déclaration de fonction faisant référence à une autre déclaration qui la suivait (le dernier exemple).

Dans les deux cas, il s'agissait de déclaration de fonctions, et le problème survenait en raison du fait que nous n'avions pas tout à fait bien implémenté l'équivalence syntaxique de typage, décrite dans la section du sucre syntaxique.

Dans les deux cas, il y avait un problème se posant dans la vérification de types de déclarations faisant référence à précédentes ou successives déclarations. Cela a été corrigé après 2 itérations en modifiant légèrement l'ordre d'instructions et la façon dont on entreprenait la mise-à-jour de l'environnement de types qui est comme suit:

```
-- Iteration 1 - Erreur dans declarations successives
infer tenv (Llet [] e) = infer tenv e
infer tenv (Llet ((vi, ei) : ds) e) =
    infer ((vi, infer tenv ei) : tenv) (Llet ds e)

-- Iteration 2 - Erreur dans declarations precedentes
infer tenv (Llet ds b) =
    infer ((map (\(v, e) -> (v, infer tenv e)) ds) ++ tenv) b

-- Iteration 3 - Aucune erreur
infer tenv (Llet ds b) =
    let
        (tenvn, tenvt) = unzip tenv
        (vars, exps) = unzip ds
        tenvn' = vars ++ tenvn
        tenvt' = (map (\e -> infer (zip tenvn' tenvt') e) exps) ++ tenvt
    in
        infer (zip tenvn' tenvt') b
```

On remarque, si l'on suit les instructions de près, comment la procédure diffère d'une itération à l'autre. La dernière itération a permis aux déclarations de faire référence à d'autre déclarations qui la suivent ou qui la précèdent.

Dans le second cas, un problème d'évaluation survenait dans le cas où une déclaration faisait référence à une déclaration successive. La solution suit de très près celle discutée pour le cas de vérification de types:

```
-- Iteration 1 - Erreur dans declarations successives
eval2 senv (Llet [] b) = (eval2 senv b)
eval2 senv (Llet ((v, e) : ds) b) =
    \venv -> ((eval2 (v : senv) (Llet ds b)) (((eval2 senv e) venv) : venv))

-- Iteration 2 - Aucune erreur
eval2 senv (Llet ds b) = \venv ->
    let
        (vars, exps) = unzip ds
        senv' = vars ++ senv
        venv' = (map evlt exps) ++ venv
        evlt = \v -> ((eval2 senv' v) venv')
    in
        ((eval2 senv' b) venv')
```

Suite à ces corrections, tous les tests du fichier ont passé.

Cependant, nous nous étions pas encore penchés sur les détails de la syntaxe du langage, surtout dans le cas d'expressions contenant des erreurs. Par exemple, notre définition initiale de la conversion d'un type `Sexp` en un `Ltype` était trop vague et laisser donc passer des syntaxes incorrects:

```
-- Version 1 - Definition trop vague
s2t' :: Sexp -> [Sexp] -> Ltype
```

```
s2t' se selist =
    case selist of
        (ta : Ssym "->" : tr : []) -> Larw (s2t ta) (s2t tr)
        (ta : Ssym "->" : tr) -> Larw (s2t ta) (s2t' se tr) -- Erreur ici
        ...

-- Version 2 - Definition rigoureuse
s2t' :: Sexp -> [Sexp] -> Ltype
s2t' se selist =
    case selist of
        (ta : Ssym "->" : tr : []) -> Larw (s2t ta) (s2t tr)
        ...
```

où l'on remarque ici que la seconde ligne dans le corps du `case` peut laisser passer des syntaxes incorrectes. Par après, la révision de l'implantation des divers `Sexp` et comment elles sont converties a été revisités et plusieurs cas similaires ont été corrigés

5 Détails sur différents choix d'implémentation

Suite à la précédente révision qui a mené à gérer différents cas de syntaxe, certains questionnements sur la syntaxe qui ne sont pas spécifiés dans la donnée du travail persistent. On tente ici d'expliquer certains de ces questionnements ainsi que nos décisions par rapport à ceux-ci.

Un premier questionnement fut celui du parenthèsage d'expressions. Pour ce qui est des expressions ou types atomiques (un nombre, une variable, un type primitif, ...) on se questionnait sur la validité des jugements suivants:

$$\text{Int} \stackrel{?}{\equiv} (\text{Int}) \qquad \text{(Type nombres entiers)}$$
$$\text{Bool} \stackrel{?}{\equiv} (\text{Bool}) \qquad \text{(Type booléens)}$$
$$n \stackrel{?}{\equiv} (n) \qquad \text{(Un entier)}$$
$$x \stackrel{?}{\equiv} (x) \qquad \text{(Une variable)}$$

Après réflexion, il nous a semblé être plus approprié d'empêcher le parenthèsage de la sorte pour les expressions atomiques puisque les parenthèses dans notre langage sont significatives $\implies$ elles indiquent l'application du constructeur de types `Scons` composés de deux `Sexp`. Ces exemples abordés étant atomiques impliquent qu'il ne s'agit que d'un `Sexp` (`Ssym` ou `Snum` selon le cas).

Un second questionnement est celui traitant sur l'expression `let` du langage. Le langage fonctionnel Haskell permet une syntaxe intéressante:

```
let in body
```

On remarque qu'aucune déclaration n'a été faite, ce qui rend l'expression `let` inutile. Est-il pertinent que notre langage adopte un tel comportement? Après avoir noté la structure du constructeur de types `Llet` pour une `Lexp`, on voit que les arguments s'agissent d'une liste de déclarations et du corps à exécuter. Rien n'empêche que la liste soit vide, car elle demeurera correctement typée. Cela nous a donc encouragé à également adopter cette approche.

Un troisième questionnement aborde le cas de construction de tuples. Par un raisonnement similaire à précédemment, on remarque que Haskell permet l'usage d'une construction particulière de tuples:

```
ghci> :t ()
() :: ()
```

Le tuple vide est une construction valide du langage. Et comme dans le cas précédent, on remarque que `Ltuple` prend une liste de `Lexp` en paramètre, ce qui n'empêche pas celle-ci d'être vide. Et bien sûr, comme démontré ci-haut, le type d'un tuple vide est tout simplement `Ltup []`. Nous avions donc décider d'adopter cette mesure également. Cette donnée semble, à première vue, inutile pour notre langage. Cela en raison du fait qu'il n'est même pas possible de la manipuler avec l'expression `fetch` puisqu'il n'y a rien à assigner. La seule utilité est vraiment peut-être de l'assigner à une autre variable ou la retourner comme résultat. Si l'on pensait à étendre le langage Psil, peut-être qu'elle s'avèrerait plus utile.

La plus grande difficulté de ce projet était d'implémenter l'inférence de types de fonctions récursives. On se rend compte que, à partir de nos jugements sur les règles de typage, qu'une limite se présente dans la règle suivante:

$$\frac{\Gamma' = \Gamma, x_1 : \tau_1, ..., x_n : \tau_n \qquad \Gamma' \vdash e \Rightarrow \tau \qquad \forall i.\ \Gamma' \vdash e_i \Rightarrow \tau_i}{\Gamma \vdash (\text{let } (x_1\ e_1)\ ...\ (x_n\ e_n)\ e) \Rightarrow \tau}$$

Cette règle explique qu'afin d'inférer le type d'une expression `let`, il faut en premier lieu étendre le contexte Γ pour qu'il contienne le type inféré des différentes déclarations (pouvant être mutuellement récursives) pour ensuite inféré le type du corps de l'expression qui correspond à celui de l'expression `let`. En ce qui concerne la récursivité mutuelle, voici certains cas à prendre en compte:

- Déclaration faisant référence à une autre qui la suit ou la précède;
- Déclaration faisant référence directement à elle-même;
- Déclaration faisant référence à une autre déclaration qui, à son tour, y fait référence.

Le premier cas a été géré dans la section sur les tests. Les deux autres décrivent dans l'un, une récursion directe pour une déclaration, et dans l'autre, une récursion indirecte d'une expression sur elle-même. Lorsque la définition d'une déclaration la ramène à se référer elle-même, notre règle de typage ci-haut ne permet pas d'inférer en un temps fini le type de l'expression de la déclaration. Quelques exemple de telles déclarations:

```
; Calcul de la factoriel - Recursion direct
(let
    (fact (n Int) Bool (if (call = n 0) 1
           (call (call * n) (call fact (call - n 1)))))
    (call fact 5))

; Exemple du pdf - Recursion indirecte
(let (even (x Int) Bool (if (call = x 0) true (call odd (call - x 1))))
     (odd (x Int) Bool (if (call = x 0) false (call even (call - x 1))))
    (call odd 8))
```

On arrive à un cas de règle indécidable puisque l'inférence de telles expressions nous ramène par la suite à inférer la même expression qui y est présente et cette boucle continue sans fin. Une possible solution que nous avions tenté d'implémenté était de poser comme hypothèse le type fournit dans le code, et puis de vérifier par après si le type du résultat obtenu correspond au type fournit. On se rend compte qu'une telle solution n'est probablement pas souhaité puisque cela détruit en quelque sorte le flux des étapes à suivre. Nous n'avions donc pas sut comment procéder pour cette section, voici la version de notre code lors de notre tentative:

```
infer tenv (Llet ds b) =
    let
        ...
        tenvt' = (map (\e -> case e of
                Lhastype _ t -> t
                _ -> infer (zip tenvn' tenvt') e) exps) ++ tenvt
...
run filename =
    ...
       let ...
           ; resT = valType val }
       in
          if resT /= ltyp
          then error ("Type mismatch: " ++ show resT ++ " != " ++ show ltyp)
          else ...
...
valType :: Value -> Ltype
valType Vnum _ = Lint
valType Vbool _ = Lboo
valType Vtuple ts = Ltup (map valType ts)
valType Vfun _ (\a -> b) = Larw (valType a) (valType b)
```

Ceci dit, l'évaluation de telles expressions se fait correctement si elles sont bien définies.

6 Notes sur la modification du code

Il n'y a pas eu de grandes modifications du code source déjà fournit. Les exceptions ont été expliquées dans les précédentes sections du document. Sinon, une autre légère modification amenée est simplement le remplacement de certains des messages d'erreurs sur la syntaxe d'expressions par des fonctions pour générer le message correspondant de sorte à alléger un peu la lecture du code.