TP2 - Rapport

Philippe Gabriel
Yan Zhuang

21 juin 2021

1 Premier aperçu

Comme première étape de travail, nous avions tenter d'apparier les données de ce travail avec celui du premier travail pratique. Nous avions donc commencer par supposer que les relations `infer` et `check` devrait ressembler à aux fonctions du même nom dans le premier travail, et que les relations `expand` et `coerce` ressemblait en quelque sorte à `s2l`. Nous avions vite remarqué que de telles suppositions étaient fausses à prendre, surtout dans la méthode d'implémentation car on remarque dans le travail sur le langage `psil` que l'on peut diviser le travail en quatre phases distinctes qui est même reflété par l'implémentation. Cependant, pour le langage μ`pts`, les relations communiquent toutes entre elles en quelque sorte. Cela a rendu la tâche assez difficile au début. La technique que l'on a employé par après pour ce travail est une sorte de "Test-driven development" (tdd). On roulait les tests à l'aide de `test_samples` en appliquant l'outil de déboguage `trace` qu'offre l'environnement de `prolog` ainsi que la relation `write` à quelques emplacements clés. Cela nous permet de situer l'emplacement des échecs d'exécution et nous permet par la suite de corriger ceux-ci.

2 Initialiser l'environnement

La première erreur qui apparaissait après le démarrage des tests était l'échec de l'inférence du type `type`. Nous regardons donc la figure de la donnée indiquant les règles de typage bidirectionnelles du langage surface et on s'apeçoit que la règle suivante semble s'appliquer à notre situation:

$$\frac{\Gamma(x) = e}{\Gamma \vdash x \Rightarrow e} \tag{1}$$

Nous avions donc rajouté une règle d'inférence supplémentaire comme-ci:

```
infer(Env, X, X, T) :-
    member(X : T, Env).
```

Cela a donc permis l'élaboration de l'environnement des types `int`, `float` et `bool`. Nous avions continué cette approche pour les autres types en ajoutant, au fur et à mesure, les règles nécessaires pour le passage de ces tests. Pour le type du `if`, il nous a été nécessaire d'ajouter un cas supplémentaire dans nos règles d'expansion comme suit:

```
expand(forall(X, B), forall(X, _, B)).
```

Cela nous a permis de faire passer le test pour `if`. Un problème est ensuite survenu pour `nil`, où on avait affair pour la première fois à une construction de `functor`. Il nous fallait la décomposer en langage interne en ajoutant un cas supplémentaire à `expand`. Il nous a été nécessaire d'écrire plusieurs versions de ce cas après plusieurs tests individuelles pour finalement arriver à cette forme:

```
expand(Fa, app(Fb, AL)) :-
    Fa =.. [N | AS],
    \+ member(N, [fun, app, arw, forall, (->), (:), let, [], (.)]),
    length(AS, L),
    L \= 0,
    functor(Fa, N, L),
    last(AS, AL),
    append(AI, [AL], AS),
    Fb =.. [N | AI].
```

Après cela, le cas particulier du type de `cons`, qui prend une liste de paramètres nous a donc inciter à modifier notre définition d'expansion d'un `forall` de sorte à l'élaborer en des `forall` currifiés:

```
% forall([x1, x2,...,xn], B) -> forall(x1, _, forall(x2, _, ...))
expand(forall(X, B), F) :-
    (X = [T | TS], TS \= [] ->
           F = forall(T, _, forall(TS, B));
           (X = [A] ->
               F = forall(A, _, B);
               F = forall(X, _, B))).
```

Suite à cette modification et à d'autres ajouts de règles de typages, l'initialisation de l'environnement était complète.

3 Test d'expressions

Après que l'environnement fut proprement initialisé, nous avions tourné notre attention au passage des tests fournis. Le premier test `sample(1 + 2).` a passé avec succès d'après le code déjà implémenté. Le second test `sample(1 / 2).` passe également, mais nous nous sommes plus tard rendu compte d'un problème se présentant après ajout des règles d'inférences portant sur la coercion. Il fut donc nécessaire d'ajouter les relations suivantes:

```
coerce(_, E1, int, float, app(int_to_float, E1)).

infer(Env, E1a, Eo, float) :-
    (infer(Env, E1a, E1b, int) ->
       coerce(Env, E1b, int, float, Eo)).
```

À ce point, nous avions décidé d'ajouter la plupart des règles de typages, des expansions du langage, ainsi que coercions restantes. Pour la majorité d'entre eux, il existe une correspondance directe entre les règles de la figure 2 et les ajouts du langage surface sous forme de sucre syntaxique ainsi que du code. Il fut nécessaire par contre de modifier un peu la définition de certains d'entre eux pour satisfaire aux différentes occurrences de code retrouvées dans les tests et qui n'étaient pas explicitement décrits dans la donnée du travail. On indique ici certaines de ces modifications apportées.

Une première est pour l'expansion du `forall` où le code gère également le cas où une liste de paramètres implicites est passée (ex.: le type de `cons`). Cette modification a déjà été expliqué dans la section précédente.

Une seconde est dans le cas de l'expansion d'une expression `let` où il a été nécessaire, dans le cas de déclarations de type `forall`, d'expliciter l'argument implicite comme suit:

```
expand(let([D = V | DS], B), LX) :-
    (D = (X : T) ->
       (T =.. [forall | _] -> % Presence d'argument implicites
           getImpArgs(T, [], PS),
           (functor(X, X, 0) ->
               (DS = [] ->
                   LX = let(X, T, fun(PS, V), B);
                   LX = let(X, T, fun(PS, V), let(DS, B)));
               X =.. [N | AS],
               convertFun(AS, V, F),
```

```
            (DS = [] ->
                LX = let(N, T, fun(PS, F), B);
                LX = let(N, T, fun(PS, F), let(DS, B))));
        (functor(X, X, 0) ->
            (DS = [] ->
                LX = let(X, T, V, B);
                LX = let(X, T, V, let(DS, B)));
            X =.. [N | AS],
            convertFun(AS, V, F),
            (DS = [] ->
                LX = let(N, T, F, B);
                LX = let(N, T, F, let(DS, B)))));
    D =.. [N | AS],
    (AS = [] ->
        (DS = [] ->
            LX = let(N, V, B);
            LX = let(N, V, let(DS, B)));
        convertFun(AS, V, F),
        (DS = [] ->
            LX = let(N, F, B);
            LX = let(N, F, let(DS,B))))).
```

Cette remarque nous a ensuite mené à réaliser que, tout comme pour `forall` où il est possible d'indiquer une liste de paramètres implicites, de même il en est pour une `fun`:

```
% fun([x1, x2,...,xn], B) -> fun(x1, fun(x2, ...))
expand(fun([], V), V).
expand(fun([A | AS], V), fun(A, fun(AS, V))).
```

Les relations `getImpArgs` et `convertFun` employées ici nous permettent de traiter des cas où il y a présence d'arguments implicites dans l'un et où il y a des paramètres directement au nom de la fonction sous forme de `functor` dans l'autre:

```
getImpArgs(arw(_, _, B), PSa, PSb) :-
    ((B =.. [forall | _]; B =.. [arw | _]) ->
        getImpArgs(B, PSa, PSb);
        PSb = PSa).
getImpArgs(forall(X, B), PSa, PSb) :-
    ((B =.. [forall | _]; B =.. [arw | _]) ->
        ((X = [_ | _]) ->
            append(PSa, X, PSa1);
            append(PSa, [X], PSa1)),
            getImpArgs(B, PSa1, PSb);
        ((X = [_ | _]) ->
            append(PSa, X, PSb);
            append(PSa, [X], PSb))).
getImpArgs(forall(X, _, B), PSa, PSb) :-
    ((B =.. [forall | _]; B =.. [arw | _]) ->
        ((X = [_ | _]) ->
            append(PSa, X, PSa1);
            append(PSa, [X], PSa1)),
            getImpArgs(B, PSa1, PSb);
```

```
        ((X = [_ | _]) ->
            append(PSa, X, PSb);
            append(PSa, [X], PSb))).

convertFun([], V, V).
convertFun([A | AS], V, F) :-
    convertFun(AS, V, B),
    (A = (X : T) ->
        F = fun(X, T, B);
        F = fun(A, B)).
```

Un autre changement apporté survient à la règle 7 de la figure 2. Comme il a été mentionné plus haut, le `forall` dans les déclarations vient ici apporter un ou des arguments implicites à la fonction. On retrouve donc ceci:

```
infer(Env, let(X, E1a, E2a, E3a), let(X, E1b, E2b, E3b), E4) :-
    check(Env, E1a, type, E1b),
    forall2arw(E1b, Eo),
    check([X : E1b | Env], E2a, Eo, E2b),
    infer([X : E1b | Env], E3a, E3b, E4).
```

Cette modification est introduite avant la vérification de `E2`. L'idée est de transformer une structure `forall` en `arw` de sorte à empêcher la règle 11 d'interférer dans un tel cas où les arguments implicites sont pris en compte:

$$\frac{\Gamma, x : e_2 \vdash e_1 \Leftarrow e_3}{\Gamma \vdash e1 \Leftarrow \texttt{forall}(x, e_2, e_3)} \tag{11}$$

La relation `forall2arw` est une simple récursion visant à éliminer les `forall` présents dans le corps d'une expression `forall` ou `arw`:

```
forall2arw(forall(X, T, Ba), arw(X, T, Bb)) :-
    ((Ba = forall(_, _, _); Ba = arw(_, _, _)) ->
            forall2arw(Ba, Bb);
            Bb = Ba).

forall2arw(arw(X, T, Ba), arw(X, T, Bb)) :-
    ((Ba = forall(_, _, _); Ba = arw(_, _, _)) ->
            forall2arw(Ba, Bb);
            Bb = Ba).
```

Finalement, le dernier changement par rapport à la donnée est le traitement relatif à la coercion du `forall`. Le code s'agit simplement de ceci:

```
infer(Env, E1, Eo, E3b) :-
    (member(E1 : forall(X, E2, E3a), Env) ->
        forallRec(Env, E1, forall(X, E2, E3a), Eo, E3b)).
```

La relation auxiliaire `forallRec` s'occupe de bien décomposer les `forall` récursifs pour leur appliquer par la suite la coercion du `forall`:

```
forallRec(Env, E1, forall(X, E2, E3a), Eo, E3b) :-
    (E3a = forall(Xi, E2i, E3ai) ->
        forallRec(Env, E1, forall(X, E2, E3ai), Eoi, E3bi),
        forallRec(Env, Eoi, forall(Xi, E2i, E3bi), Eo, E3b);
```

```
        subst(Env, X, _, E3a, E3b),
        coerce(Env, E1, forall(X, E2, E3a), E3b, Eo)).
```

Cela vient conclure les détails de l'implémentation et, à ce stade, le langage semble avoir été bien et complètement implémenté et tous les tests passent comme prévues.

4 Tests d'expressions avancées

Nous nous intéressons maintenant à des expressions un peu plus complexes que celles offertes dans le code. Voici un exemple de la fonction `double` abordée en Haskell, une fonction d'ordre supérieur en μ`pts`:

```
sample(let([double : forall(t, arw(o, (t -> t -> t), (t -> t))) =
    fun([f, x], f(x, x))], double((/), 5.5))).
```

Un tel test a permis de développer plus en profondeur les relations `forall2arw` et `getImpArgs` de sorte à être en mesure de passer de tels tests encore plus complexes. À notre surprise, ce test a bien passé, en inférant le type de cette expression à être `float` tel qu'attendu.

5 Notes sur la modification du code

Il n'y a eu aucune de modification au code source fournit à l'exception de déplacement de code dans certains cas pouvant être utile pour respecter la limite de 80 caractères par ligne.