Functional and logic programming lab 8th report

Yoshiki Fujiwara, 05-191023

課題 1 から課題 4 を kadai1 の folder に、課題 5 を kadai5 の folder に、課題 7 を kadai7 の folder に実装した。

1 Q.1: 再汎単一化子

1.1 解答

(1) $\alpha = Int, \beta = int \to Int$
(2) 解なし
(3) $\alpha = Int, \beta = Int$
(4) $\alpha_1 = \beta_1 \to \beta_2, \alpha_2 = \beta_1 \to \beta_2, \alpha_3 = \beta_1 \to \beta_2$
(5) 解なし

2 Q.2: ty_subst の定義

2.1 実行例

この実行例は、tySyntax.ml と constraintSolver.ml のファイルの中身を ocaml 上で実行させることで確認を行った。

Code 1 動作例

```
# ty_subst [(1,TyInt); (2,TyVar(4))] TyInt;;
- : ty = TyInt

# ty_subst [(1,TyInt); (2,TyVar(4))] TyBool;;
- : ty = TyBool

# ty_subst [(1,TyInt); (2,TyVar(4))] (TyVar(2));;
```

```
- : ty = TyVar 4

# ty_subst [(1,TyInt); (2,TyVar(4))] (TyFun(TyVar(2), TyInt));;
- : ty = TyFun (TyVar 4, TyInt)

# ty_subst [(1,TyInt); (2,TyVar(4))] (TyFun(TyVar(2), TyVar(1)));;
- : ty = TyFun (TyVar 4, TyInt)
```

2.2 考察

まず、型を定義する。型の定義は前回の mli file と同様に行った。また、newvar については新しい関数が定義されるたびに increment して、変数の通し番号がかぶら内容にするため、tyvar は int 型で定義した。

ty_subst 型の関数の働きは、sigma と t を受け取って、t がもし、sigma という代入するリストの中に入っていれば、代入を行うという働きである。よって、t が ty 型のなにかによって場合分けをする。この動きはスライドの型代入の部分に詳しく記載されている。t が VInt や VBool であった時はそのまま返す。

t が変数であった場合は、その変数に対して代入が起こるか否かを sigma を探索して確認する。そのための関数が、lookup である。lookup によって、見つかった場合は代入を行い、見つからなかった場合は元の変数をそのまま返す。

t が関数であった場合には、関数の引数、返り値のそれぞれについて ty_subst を行えば良い。

3 Q:3 代入を合成する関数

3.1 動作例

Code 2 動作例

```
# let sig1 = [(1,TyInt);(2,TyVar(4))];;

# let sig2 = [(5,TyBool);(9,TyVar(1));(3,TyBool);(6,TyFun(TyInt,TyBool))];;

# compose sig1 sig2;;
- : (tyvar * ty) list =
[(1, TyInt); (2, TyVar 4); (5, TyBool); (9, TyInt); (3, TyBool);
```

```
(6, TyFun (TyInt, TyBool))]
```

3.2 考察

compose 関数を定義する。compose 関数の大きな流れとしては、sig1 と sig2 を受け取り、sig2 の代入した後について、sig1 の代入を行うという、list_subst 関数を定義する。そして、その関数を適用した結果を substituted として、sig1 の代入前が substituted の代入前と被っていたら削除するという関数、list_erase を定義する。

まず一つ目の関数は、list_subst 関数である。この関数は、list に対して substitution を行う関数である。ty_subst では、ty 型に対してしか代入操作が行えなかったため、sig2 を match 文で分解する。sig2 で $\alpha \to \beta$、sig1 で $\beta \to \gamma$ となっているものを $\alpha \to \gamma$ にする操作であるので、それ通りに実装する。

その次の関数は、list_erase である。この関数は、sig2 と sig1 の競合をなくす関数である。この操作は list_subst 関数を行ったあとでは必要ないが、一応実装した。この関数のために first_check という関数を定義した。これは、先頭要素を順に見て行って存在すれば true, 存在しなければ false を返す関数である。この関数を用いて、substituted の中に $\alpha \to \beta$ があり、sig1 の中に $\alpha \to \gamma$ がある時の、$\alpha \to \gamma$ を削除する作業を行う。

4 Q:4 単一化を行う関数

4.1 動作例

Code 3 動作例

```
unify [(TyVar(1),TyInt);(TyVar(2), TyFun(TyVar(1),TyVar(1)))];;
- : (tyvar * ty) list = [(2, TyFun (TyInt, TyInt)); (1, TyInt)]

unify [(TyInt,TyFun(TyInt,TyVar(1)))];;
Exception: TyError.

unify [(TyFun(TyInt,TyInt),(TyFun(TyVar(1),TyVar(2))))];;
- : (tyvar * ty) list = [(1, TyInt); (2, TyInt)]

unify [(TyFun (TyVar(1),TyVar(2) ),(TyFun(TyVar(2),TyVar(3) )));
(TyVar(3), TyFun (TyVar(4),TyVar(5)))];;
- : (tyvar * ty) list =
[(3, TyFun (TyVar 4, TyVar 5)); (1, TyFun (TyVar 4, TyVar 5));
```

```
(2, TyFun (TyVar 4, TyVar 5))]

unify [(TyFun (TyVar(1),TyVar(1)) , TyFun (TyVar(2), TyVar(3)));
(TyVar(3), TyFun(TyInt,TyVar(2)))];;
Exception: TyError.
```

これは Q.1 の結果と等しくなっており、正しい。

4.2 考察

まず、前回も実装した union 関数を定義した。そのために、setin 関数も定義しているが、この関数は union 以外で使わないため、説明は割愛する。union 関数は、和集合をとる関数である。

次に include_check 関数を定義した。これは、スライドの単一化アルゴリズムにおける、t は alpha を含まないにあたる部分である。含んでいればエラーが出るようにしている。

subst 関数は、ty_subst 関数はリストに対する代入が行えないため定義した関数である。

本題の unify 関数に入る。unify 関数は再帰関数で記述する。const には制約が入り、制約の一つ一つについて見ていく。

Int = Int や Bool = Bool は自明であるので覗く。

関数がきた場合は、引数同士、返り値どうしが等しいという式に書き直し、残りの制約と union をとる。

変数が来た場合は、スライド通りに、compose 関数と subst 関数を用いて評価を行う。この時 t は、include_check を行う必要がある。そこでエラーが出た場合は raise error すれば良い。

以上の形におさまらないものについては error を raise すれば良い。

5 let 多層

5.1 実行例

Code 4 動作例

```
# let f = fun x -> x in if f true then f 1 else 3;;
int
- = 1
# fun f -> fun x -> let g = f in if g true then g 1 else 3;;
Error = Unmatched type
```

```
# let apply = fun f -> fun x -> let g = f in if g true then g 1 else 3;;
Error = Unmatched type
# let rec apply x = let g = apply in if g true then g 1 else 3;;
Error = Unmatched type
```

5.2 考察

5.2.1 変数

まず、infer_expr の変数の処理についてかく。まず型環境を見て、型変数に対応する型スキームを取ってくる。これについて、instantiate という関数を実行する。instantiate 関数は、bind されている変数に新しい変数を割り当てて、それぞれに代入する操作を行う関数である。create_substitution 関数で、新しく作った変数への割り当てを行う代入関数を作っている。

5.2.2 let in 文

let in 文についてはスライド通りの流れで実装を行なった。

let in 文のスライドを実装するにあたって必要になった関数の説明を順にしていく。

まず、get_type_vars 関数の実装を行なった。これは ty 型から変数のリストを受け取る関数である。

次に、comp_list_list 関数の説明をする。この関数は、リストを二つ受け取り、(li1,li2 とする)、li1 に含まれていて、li2 に含まれていない要素のリストを返す関数である。

$\Delta = \Gamma\sigma$ をするにあたり、env_subst という関数を実装した。ここで代入を行うのは、$\forall$ で bind されていないものについてであることに注意する。Γ 一つ一つに対して、代入を行なっていくために、一つに対しての代入を行う関数、env_subst_hitotu を定義した。その関数の中では、まず、Γ の要素のうち、代入されるべきもののリストを lets_substitute にいれている。sigma をその代入するべき変数のみにしたものを sigma2 とする。この操作をしないと、bind されている変数についても代入が行われてしまう可能性がある。そしてそれを用いて代入を行えば良い。

次に、スライドの P を定義する。P は s1 に現れて、Δ に現れない型変数の集合である。一方の list に含まれていてもう一方に含まれていない要素の list は comp_list_list を用いてかけるので、comp_list_list を用いてかく。この時、s1 に含まれる変数は、get_type_vars で list として受け取ることができる。create_p で P に含まれる自由な変数を取ってきて

いる。

create_P について、これは型環境から変数のリストを取ってくる関数である。よって一つ一つの scheme について、pick_var_from_schema という schema から変数を取ってくる関数を用いて、変数のリストを取ってきている。

これによって generalize によって、$\forall P.s_1$ が作ることができた。

スライド通りに newtenv を上で作った generalize を用いて定義して、v2 を評価すれば、let in 文については完成する。

あとは型環境を (name * type_schema) になるように型が整合しない部分を変えていけば完成する。

6 Q.6: Occurance check

出現検査がなければ変数が無限の項を持たなければならなくなり、項は有限と仮定したことに矛盾する。その例を以下に挙げる。

$\{\alpha = int \to int \to \alpha, \beta \to \alpha \to int\}$ を unification すると、

$\{\alpha = int \to int \to \alpha, \beta = int \to int \to \alpha \to int\}$ となり、それをもう一度 unification すると、

$\{\alpha = int \to int \to \alpha, \beta = int \to int \to int \to int \to \alpha \to int\}$ となる。

これは確かに、unification を繰り返すことで無限の項となってしまう例である。

出現検査をしないで単一化することは、cyclic structures を考えていることに相当する。

7 Q.7: 繰り返し評価

7.1 動作例

Code 5 動作例

```
# let x = fun x -> (x,x) in
let x = fun y -> x (x y) in
let x = fun y -> x (x y) in
let x = fun y -> x (x y) in
let x = fun y -> x (x y) in
let x = fun y -> x (x y) in
x (fun x -> x);;
```

```
a39->a39
- = 2*2*2*2*2*2*2*2*2*2*2*2*2*2*2*2*2*2*2*2*2*2*2*2*2*2*2*2*2*<fun>

# let x = fun x -> (x,x) in
let x = fun y -> x (x y) in
let x = fun y -> x (x y) in
x (fun x -> x);;
a18->a18
- = 2*2*2*<fun>
```

7.2 考察

まず、今回の型推論の難しいポイントとして、型推論が終わりきらないということがある。これが原因となって、型推論や評価が終わらないということになってしまっている。今回は同じ expression を評価する際に、片方だけを評価するという方法を取ることで、評価・推論の時間を削減した。まず、そのために、考えられる方法としては、一定長のリストに評価したものの結果をためておく方法と、Pair の両辺に同じ expression がきたら、評価を一回だけにする方法が考えられる。前者の方が一般性を保持しているが、今回の課題要件を満たすには後者で十分であるため、後者で実装した。

今回の実装では eval_expr で繰り返し (同じ expression) の起こった回数を数えて、infer_expr では繰り返しの最小単位を返すという実装を行った。型推論の部分では、繰り返しの最小単位の型を返せば良い。そのために VRepeat という新しい型を導入した。infer_expr の EPair の部分で、pair の前後の入力が等しければ、TyRepeat という型に入れる。また、constraintSolver の部分もそれに応じて変更した。これによって、型の出力は一つ分になる。

また、eval_expr の部分で、繰り返し回数を数える。新しく value の variant に VRepeat という新しい型を追加する。新しい型には繰り返す型と繰り返す回数が記載されている。ここで syntax.ml の print_value での出力の際繰り返されている回数分 2*で出力すれば良い。そうすることで、実際であれば console 内に収まらない型を省略して書くことができる。

8 Q.8: 論文

Conor McBride, "First-order unification by structural recursion", Journal of Functional Programming, 13(6), 1061—1075, 2003 の要約と感想

8.1 要約

この論文は、単一化アルゴリズムの停止性を証明する方法として、従来と異なる方法を、データの型を工夫することで実現している。ここでは、データ型を Inductive なデータにすることで構造帰納法を用いれるようにしている。この論文での構造帰納法とは、型を階層化して、Term n と記述されている、(a set of n variables) を表すものに関する帰納法を用いることである。筆者は、自身の定義した datatypes の deduction style な記述方法で、match 文などの基本構文の記述を説明し、その後、substitution,Occurance check, unification について説明している。

8.2 感想

単一化アルゴリズムの停止性の証明は別のサイトに書いているようであったが、そこまで読むことができなかったため、また機会があれば読んでみたいと感じた。