計算機科学実験 3SW レポート 3

学籍番号: 1029350386
氏名: 高橋奏太

2025 年 7 月 6 日

ex4.2.1(必修)

1 設計方針

Miniml2 用の型推論を実装するためにまず syntax.ml に Miniml の型を表す型 ty とそれを扱う関数を追加する。

cui.ml には現環境だけでなく型環境も渡してから、型推論後、式評価の流れを取るように変更。

main.ml では REPL の最初の呼び出しで大域環境の型環境も渡すように変更。

typing.ml では syntax.ml で定義されているデータ型を受け取ったら現在の型環境をもとに型推論する関数を追加し、型推論アルゴリズムを実装。

2 実装にあたり工夫した点

変更にあたって typing.ml で実装した ty_decl が評価した型と型環境を返すので、その形を合わせるために typingTestGenerator.ml の中の typing 関数の形式を ty から (ty,nex_env) に変更した。

また ex3.2.3 で&&,||を追加しているのでその演算子にも対応できるように ty_prim を変更した。

3 インタプリタ動作確認

```
# -5;;
val - : int = -5
# 3 + 7;;
val - : int = 10
# 8 * (-10);;
val - : int = -80
# 4 < 2;;
val - : bool = false
# true;;
val - : bool = true
# if 2<4 then 3+5 else 6*4;;
```

```
val - : int = 8
# let x = 5;;
val x : int = 5
# let y = 8 in x + y;;
val - : int = 13
# if true then 3 else false;;
Fatal error: exception Miniml.Typing.Error("then and else branches must have
    the same type")
```

整数、真偽値、op+,*,<、条件分岐、let 宣言、let 式の型を確認している。条件分岐においては then 節と else 節の型が違うならエラーを出せていることも確認できている。

ex4.3.1(必修)

1 設計方針

まず、string_of_ty 関数は、TyVar 以外は型の名前を返したり再帰的に string_of_ty 関数を用いればよい。TyVar に関しては引数の n が 0 なら'a、1 なら'b を表すように補助関数 string_of_tyvar を追加。

次に freevar_ty 関数は、与えられた型 ty の中にある型変数の集合を返す必要があり、TyVar,TyFun のときに MySet モジュールの singleton 関数や union 関数を用いて集合を返している。

2 実装にあたり工夫した点

エラーを回避するために typing.ml で subst_type と unify を一旦関数として成立させることで runtest を成功させた。以降の必修課題でこの 2 つの関数を改良していくこととなる。

3 動作確認

```
string_of_ty

utop # string_of_ty (TyInt);;
- : string = "int"

utop # string_of_ty (TyBool);;
- : string = "bool"

utop # string_of_ty (TyFun (TyFun (TyVar 0, TyInt), TyVar 30));;
- : string = "(('a -> int) -> 'e1)"

freevar_ty

utop # to_list(freevar_ty (TyVar 1));;
- : int list = [1]
```

```
utop # to_list(freevar_ty (TyInt));;
- : int list = []

utop # to_list(freevar_ty (TyFun (TyVar 10, TyBool)));;
- : int list = [10]
```

ex4.3.2(必修)

1 設計方針

subst のリストの中身を左から順番に適用して変化させた型を毎回保持する必要があるので、subst の中身 1 つを受け取り型 ty を変化させる再帰的な補助関数を作り、その補助関数を let in 式で List.fold_left を用いてリストの左から順番に適用させ型の変化を累積値として保持していく。

2 実装にあたり工夫した点

関数の中身が複雑に見えるので、型注釈を加えることで安全性を増した。また、型が基本型の時は新しいオブジェクトを作成せずそのまま返す選択をとることで処理性をあげた。

3 動作確認

```
utop # let alpha = fresh_tyvar() in

subst_type [(alpha, TyInt)] (TyFun (TyVar alpha, TyBool));;

- : ty = TyFun (TyInt, TyBool)



utop # let alpha = fresh_tyvar() in

let beta = fresh_tyvar() in

subst_type [(beta, (TyFun (TyVar alpha, TyInt))); (alpha, TyBool)] (TyVar
    beta);;

- : ty = TyFun (TyBool, TyInt)
```

1 つ目の実行過程を説明すると、subst として置換リスト [(alpha, TyInt)] を ty として (TyFun (TyVar alpha, TyBool)) を受け取り、List.fold_left で初期値 (TyFun (TyVar alpha, TyBool)) が acc_ty に渡され subst の 1 番左の要素 (今回は要素が 1 つだけ) の (alpha, TyInt) が subst_pair に渡される。この 2 つを用いて補助関数 apply_subst を適用し再帰的な処理もはさむことで、結果を返す。もしここで subst の中身が複数あれば今の 1 回目の結果をもう一度 acc_ty に渡しリストの 2 番目の要素を適用させる、と

いう流れになる。

ex4.3.3(必修)

1 設計方針

unify 関数は、制約に基づく型推論において制約解消の部分を担っている。設計の方針としては、等式制約のリストの中身を左から順に取り出し、パターンマッチによって処理を行っている。それぞれのパターンでの処理は unify の定義に基づき行う。その際、新しく発生した型代入をまだ単一化できていない等式制約に適用させる関数や、オカーチェックを行う関数など補助関数を利用することで unify 関数を設計した。

2 実装にあたり工夫した点

レポート 2 にてリスト表記に MiniML を対応させたので等式制約がリストの場合にもリストの中身に対して再帰的に制約を考えるというパターンマッチで対応させた。

型の等式集合に型代入を適用する subst_eqs 関数では ex4.3.2 で実装した subst_type を List.map を使うことで eqs のすべての要素に適用して目的を実現できている。

オカーチェックを行う関数では ex4.3.1 で実装した freevar_ty を使うことで出現している型集合を求め、それを MySet モジュールの member 関数を適用させて alpha が型に含まれるかの確認を実装した。

また、全体で型注釈をはさみ、安全性を確保した。

さらに、パターンマッチの順番を同一の型の後ろに型変数とそれ以外の型を置くことで alpha=alpha という求めていないケースを省くこととしている。

3 動作確認

```
utop # unify [(TyFun (TyVar 0, TyVar 1), TyFun (TyVar 1, TyVar 0))];;
- : (int * ty) t = [(0, Miniml.Syntax.TyVar 1)]

utop # unify [(TyFun (TyVar 0, TyBool), TyFun (TyInt, TyVar 1))];;
- : (int * ty) t = [(0, Miniml.Syntax.TyInt); (1, Miniml.Syntax.TyBool)]

utop # unify [(TyVar 0, TyFun (TyInt, TyVar 0))];;
Exception: Miniml.Eval.Error "Unification Error: Occur check failed".
```

dune utop にて open Miniml__Syntax と open Miniml__Typing を行って動作確認を実行した。上から、型変数を別の型変数に単一化、型変数を具体的な型に単一化、おかーチェックにより失敗、のケースである。

ex4.3.4(必修)

1 考察

単一化アルゴリズムでは、型の等式制約 $\alpha = \tau$ または $\tau = \alpha$ を処理する際に、型変数 α を型 τ に単一化 (置き換え) する。もし τ の中に型変数 α 自身が含まれている場合（$\alpha \in \mathrm{FTV}(\tau)$）にオカーチェックを行わずに単一化を許容すると、$\alpha$ は自分自身を含む型に置き換えられてしまう。例えば、制約が $\alpha = \mathrm{int} \to \alpha$ であった場合、オカーチェックがなければ、α は $\mathrm{int} \to \alpha$ に置き換えられ、その中の α がさらに $\mathrm{int} \to \alpha$ に置き換えられというように、$\mathrm{int} \to (\mathrm{int} \to (\mathrm{int} \to ...))$ のような無限に展開される型が生成されてしまう。その場合、無限ループとなり、この単一化アルゴリズムが有限時間で停止せず決定不能な問題となってしまう可能性があるから、オカーチェックの条件は必要だと考えられる。

ex4.3.5(必修)

1 設計方針

ex4.3.3 で unify 関数を実装する際に補助関数として subst_eqs 関数を実装済みだったので、それ以外の型代入を型の等式集合に変換する関数 eqs_of_subst を完成させる。

また、ty_prim と ty_exp に関しては型だけでなく、制約集合や型代入を返すように変更を加える。それに伴い、ty_decl に型エラーが生じるので変更を加えることで、型推論できるようにする。

2 実装にあたり工夫した点

eqs_of_subst において、与えられた subst の 1 つ目から型等式に変換するために subst_type を使って型代入の変換先の型が最終的にどの型になるかを求めたのちその型を等式の相手にすることで制約を確実に満たす変換となっている。

また、ty_prim では返り値として制約集合と型を渡しているので、ty_exp 内では、BinOp の時に等式制約の集合を ty_pim を利用することで得たのち連結させてから unify で単一化することで最終的な型代入と返り値の型を得ている。

このやり方 (制約集合の形で連結してから単一化するという 2 ステップ) によって部分式に型変数が含まれていても問題なく解決できるようになっている。

IfExp の実装では、If 式の返り値が不明なため一旦 TyVar(fresh_tyvar()) と型変数で置き後から求めている。その際、各部分式の等式制約の集合と If 式の制約 (条件式は Bool 型、Then 節と Else 節は同じ型で If 式の返り値の型と同じ) を連結させてから単一化し、型を解いている。

LetExp では ty1 にまだ未解決の型変数がある可能性があるので環境を追加し ty2 を評価したのち制約として単一化するという基本の順番を守っている。

AppExp では制約として関数式の型は引数式の型から結果の型 (型変数で一旦置いた) への関数型というものを追加して考えた。

更に、ty_exp の返り値が変わったのでそれに伴い ty_decl を修正することにした。この課題では Miniml3 に対応する型推論アルゴリズムを作ればよいので、パターンマッチの内、Exp と Decl のときを変更させた。ただ、型代入の返り値は使わないのでワイルドカードで受け取り捨てている。

3 動作確認

```
# fun x -> x+1;;
val - : (int -> int) = <fun>
# fun f -> fun x -> f x;;
val - : (('c -> 'd) -> ('c -> 'd)) = <fun>
# let fact = fun x -> x*x;;
val fact : (int -> int) = <fun>
# if true then 10 else 20;;
val - : int = 10
```

関数式、関数適用式、let 宣言、if 式の型推論を確認している。

レポート 2 フィードバック

レポート 2 のフィードバックで parser.mly に shift/reduce conflict があるとのことだったので、修正を加える。menhir –explain src/parser.mly コマンドで確認してみると、エラーとして非終端記号の型が明示されていない状況が、警告として入力トークンをシフトするかリデュースするかの競合が出されている。

まず、エラーをなくすため、LetExpr などの Syntax.exp の型と、リストで用いている非終端記号 Syntax.exp list などの型を %type で明示する。その際、警告を回避するための LET_DEF も新たに追加するので型宣言しておく。

次に警告として出されている競合は、SEMISEMI の競合と RARROW の競合がある。1 つ目の SEMISEMI の競合では、let x = 1;; の入力がトップレベルのシフトか、let_sequence のリデュースかがわからないので、回避するために let_sequence の定義を let 宣言が 2 つ以上並ぶように変更して競合を防いだ。また、2 つ目の RARROW の競合では、param_list が単一の ID も許容するため Fun x RARROW e という入力が x を ID とみて単一引数関数とみるか、x を param_list とみて複数引数関数とみるかがわからないので、どちらも param_list を経由した複数引数関数として param_list が直接 ID のときに単一引数関数としてみるように変更して競合を解消した。

実験の感想

このソフトウェア実験を通して、そもそものプログラミング言語がどのようなステップで機能を増やしていったかを追随することで深い理解を得ることができた。3 章の MiniML4 を作成する段階でバックパッチを用いてダミーの環境を作ってから一旦環境を拡張してしまうような手法があったが、4 章の型推論でも fresh_tyvar で型変数を作り一旦当てはめてから等式制約を結合し単一化することで最終的な型と

型代入を得るやり方には 3 章の手法と似たような考え方を感じた。全体を通して、言語を作る上での考え方や、型推論でのアルゴリズムや補助関数同士のつながりなど驚くべきステップが多くあった。実験を通じて自身が普段使っているプログラミング言語というものへの、今までなかった視点からの理解が深まったと思える。