レポート3（型推論実験）

Name: Pyii Phyo Maung
Student ID: 1029322149

August 6, 2023

1 Exercises Done

ex4_2_1
ex4_3_1
ex4_3_2
ex4_3_3
ex4_3_4
ex4_3_5
ex4_3_6
ex4_4_1
ex4_4_2
ex4_4_4

1.1 Description of Exercises

Exercise 4.2.1 [必修]
MiniML2 のための型推論アルゴリズムを実装するためにコードに加えるべき
変更を以下に示す．これを参考にしつつ，上記のT-Int
とT-Plus
のケースにならって，すべての場合について型推論アルゴリズムを完成させ
よ．また，インタプリタに変更を加え，型推論ができるようにせよ．

Exercise 4.3.1 [必修]
string_of_ty と freevar_ty を完成させよ．string_of_ty は ty -> string 型
の関数で，ty 型の値を受け取るとその文字列での表現を返す．TyVar 0 が "'a", TyVar 1 が "'b" の
ように，OCaml での型変数の文字列表現と合わせること．freevar_ty は，
与えられた型中に現れる型変数の集合を返す関数で，型は ty -> tyvar MySet.t と
する．型 'a MySet.t は mySet.mli で定義されている型 'a の値を要素と
する集合を表す値の型である．

Exercise 4.3.2 [必修]
型代入に関する以下の型，関数を typing.ml 中に実装せよ．

```
type subst = (tyvar * ty) list

val subst_type : subst -> ty -> ty
```

例えば，

```
let alpha = fresh_tyvar () in
subst_type [(alpha, TyInt)] (TyFun (TyVar alpha, TyBool))
```
の値は `TyFun (TyInt, TyBool)` になり，

```
let alpha = fresh_tyvar () in
let beta = fresh_tyvar () in
subst_type [(beta, (TyFun (TyVar alpha, TyInt))); (alpha, TyBool)] (TyVar beta)
```
の値は `TyFun (TyBool, TyInt)` になる．

Exercise 4.3.3 [必修]
上の単一化アルゴリズムを `(ty * ty) list -> subst` 型の関数 `unify` として実装せよ．

Exercise 4.3.4 [必修]
単一化アルゴリズムにおいて，オカーチェックの条件 α`FTV(`τ`)` はなぜ必要か考察せよ．

Exercise 4.3.5 [必修]
他の型付け規則に関しても同様に型推論の手続きを与えよ(レポートの一部としてまとめよ)．そして，以下の `typing.ml` に加えるべき変更の解説を参考にして，型推論アルゴリズムの実装を完成させよ．

Exercise 4.3.6 [**]
再帰的定義のための `let rec` 式の型付け規則は以下のように与えられる．

Γ`,f:`τ`1→`τ`2,x:`τ`1e1:`τ`2`Γ`,f:`τ`1→`τ`2e2:`τ`  `Γ`let rec f=fun x →e1 in e2:`τ`T-LetRec`

型推論アルゴリズムが `let rec` 式を扱えるように拡張せよ．

Exercise 4.4.1 [**]
多相的 `let` 式・宣言ともに扱える型推論アルゴリズムの実装を完成させよ．

Exercise 4.4.2 [*]
以下の型付け規則を参考にして，再帰関数が多相的に扱えるように，型推論機能を拡張せよ．

`Exercise 4.4.4 [**]`
型エラーが起こった際にエラー箇所が指摘できるように実装を改善せよ.

1.2 Necessary Explanations from Feedback

1.2.1 Exercise 4.3.4

The occurs check in the unification algorithm is a crucial step in preventing infinite types. In the context of type inference, a unification algorithm is used to determine a substitution that makes two types identical.

The occurs check is a condition in the unification algorithm that checks whether a type variable α occurs in a type τ. If α does occur in τ, attempting to substitute α with τ would lead to a circular or infinite type, which is not allowed.

For example, consider the situation where α occurs in τ. If we didn't perform the occurs check and allowed $\alpha = \tau$, it would imply that $\alpha = \alpha \to \alpha$, which further unrolls to $\alpha = \alpha \to \alpha \to \alpha$, and so forth. This results in an infinite, non-terminating type, which is not feasible in practice.

Therefore, the occurs check is necessary to prevent such circular or infinite types from forming, maintaining the soundness of the type inference algorithm.

2 プログラムの設計原則と詳細な説明

2.1 cui.ml

Command User Interface (CUI) ファイルは、コマンドラインを通じてユーザとの対話を担当します。

設計原則：主な設計原則は、インタラクションのためのREPL (Read-Eval-Print Loop) を提供することです。プログラムは入力を読み取り、それを評価し、結果を出力します。実行時エラーをキャッチしてユーザに意味のあるメッセージを表示するための例外処理が使用されています。

実装：実装は直感的で、ユーザからの入力を継続的に読み取り、評価器に渡し、結果を出力するループがあります。必要に応じて例外をキャッチして出力します。

2.2 environment.ml と environment.mli

これらのファイルは、環境データ構造を定義します。

設計原則：環境は評価と型推論の重要な部分であり、それは変数名をその値または型にマッピングします。環境はペアのリストとして設計されており、これは環境を容易に走査して更新することができます。

実装：環境は（変数、値/型）ペアのリストとして実装されています。環境を拡張する（新しい変数-値/型ペアを追加する）、環境を適用する（変数の値/型を取得する）、および2つの環境をマージする関数があります。

2.3 eval.ml

このファイルは評価器を実装します。

設計原則：評価器は評価のためにビッグステップセマンティクスを使用して設計されています。これは、式の値を直接計算します。評価器は、OCamlの強力な機能であるパターンマッチングを使用して、式の構造をパターンに一致させ、式の形を決定します。

実装：‘eval‘ 関数は、式と環境を取り、その形に基づいて式を評価します。変数、関数、関数の適用などのさまざまな種類の式に対するケースがあります。

2.4 mySet.ml と mySet.mli

これらのファイルは、セットの操作を定義します。

設計原則：これらのファイルは、型推論アルゴリズムで自由変数を追跡するために使用されるセットを実装します。セットはソートリストとして実装され、これはセットを実装するためのシンプルで効率的な方法です。

実装：セットはソートリストとして実装されています。セットを作成する、要素がセットのメンバーであるかどうかを確認する、2つのセットの和を取る、2つのセットの差を見つける、などの関数があります。

2.5 parser.mly

このファイルは、言語の構文とその解析方法を定義します。

設計原則：このファイルは、言語の構文を記述するための文脈自由文法を使用して設計されています。パースと評価の分離は、ここでの主要な設計原則であり、コードをよりモジュール化し、理解しやすく、保守しやすくします。

実装：パーサは、OCamlのMenhirパーサジェネレータを使用します。それは、文脈自由文法を使用して言語の構文を定義し、セマンティックアク

ションを使用して解析されたトークンから抽象構文木（AST）ノードを構築します。

2.6 syntax.ml

このファイルは、言語の抽象構文を定義します。

設計原則：抽象構文は、OCamlのデータ型を使用して定義され、プログラムを表現するための型安全な方法を提供します。また、構文を扱うためのユーティリティ関数も含まれています。

実装：言語の構文は、OCamlのデータ型として定義されています。ユーティリティ関数には、二項演算を文字列に変換する、新しい型変数を生成する、などが含まれています。

2.7 typing.ml

このファイルは、型推論アルゴリズムを実装します。

設計原則：このアルゴリズムは、Hindley-Milner型システムの形を使用します。これは、型推論をサポートする多相型システムです。このアルゴリズムの背後にある設計原理は、式の構造と現在の型環境に基づいて型方程式のセットを生成し、これらの方程式を解くことで、式の型を推論することです。

実装：型推論アルゴリズムは、環境と式を取り、式の型を推論する‘infer‘関数を使用します。それは、異なる形の式に対して型方程式を生成し、これらの方程式を解くために単一化関数を使用します。

2.8 lexer.mll

このファイルは、言語の字句解析器を定義します。

設計原則：字句解析器は、入力ストリームをトークンに分解します。これにより、文法解析器がこれらのトークンを使用して抽象構文木を構築できます。字句解析器は、正規表現を使用してトークンを識別します。

実装：字句解析器は、OCamlのlexingライブラリを使用して実装されています。それは、トークンの正規表現と、それらを認識したときに何をすべきかを定義します。

2.9 main.ml

このファイルは、プログラムのエントリーポイントを提供します。

設計原則：主な目標は、プログラムを簡単に起動して実行できるようにすることです。それはCUIを開始し、ユーザとのインタラクションを開始します。

実装：このファイルは非常にシンプルで、CUIモジュールの‘main‘関数を呼び出すだけです。

2.10 mySet.ml と mySet.mli

これらのファイルは、集合の操作を定義します。

設計原則：このファイルは、型推論アルゴリズムで自由変数を追跡するために使用される集合を実装します。集合はソートリストとして実装され、これは集合を実装するためのシンプルで効率的な方法です。

実装：集合はソートリストとして実装されています。集合を作成する、要素が集合のメンバーであるかどうかを確認する、2つの集合の和を取る、2つの集合の差を見つける、などの関数があります。

2.11 独自の実装

このプロジェクトでは、OCamlの強力な機能を活用して、型安全な抽象構文の定義、パターンマッチングを使用した評価と型推論、型エラーをキャッチしてユーザに報告するなど、いくつかの革新的な実装が行われています。

3 感想（難しかった点や実験への要望）

この実験は非常に困難であり、挑戦的でしたが、それが私に多くの知識と洞察をもたらしたため、楽しみながら取り組むことができました。私はOCamlを使用した開発の複雑さを理解し、その中で重要な経験を積むことができました。

Duneを使用すると、特に同時に複数のテストを実行しているとき、ターミナルでの結果が見づらくなることに気づきました。テストの数は、エラーが発生しない限りは表示されません。それは単に「XX秒でXXテストを実行しました」と表示されます。エクササイズの番号が含まれていると、どのテストが成功し、どのテストが失敗したのかを追跡するのがより容易になります。この情報は、テストの結果を解析する際に非常に有用です。確かに、これはDuneのテストファイルを変更することで実現できますが、それは時間がかかる作業です。

一方、Dune runtestによるエラーキャッチングは非常に効果的だと思いました。しかし、Menhir Parser Basics Errorなどの一部のエラーでは、それが不十分であると感じました。エラーメッセージにはもっと詳細と提案が含まれていてほしいと思いました。これにより、問題の原因を特定し、それを解決するための具体的なステップを計画するのがより簡単になります。エラーメッセージがより具体的であれば、より効率的に問題を解決でき、結果的にはより高品質なコードを書くことができます。

以上のような挑戦にもかかわらず、この実験は私にとって非常に有益であると感じています。それは私に、大規模なソフトウェアプロジェクトにおけるテストとデバッグの重要性を理解させ、問題を解決するための新たなスキルと戦略を獲得させました。