プログラミング初学者のためのプログラミング環境の構築

石井柚季　(指導教員：浅井健一)

1　はじめに

強い型付き言語を使う利点として,「とりあえずコンパイラの言うとおりに型を合わせていけば実行可能なプログラムができあがる」ということが挙げられる．強い型付けがユーザにとっていい意味で「制約」となっているのである．そのため強い型付き言語はプログラミング初学者に適した言語であると考えられる．また，本学科には OCaml を使ったプログラミングの授業が学部生向けに毎年行われている．

しかし，強い型付けや，プログラミングに慣れていない初学者にとってこの「制約」は「難しい概念」として捉えられてしまう場合がある．そこで本論文では本学科の関数型言語の授業で使用されることを目的に，他の強い型付けを利用したプログラミング環境を参考にしながらプログラミング初学者のためのプログラミング環境を Emacs の拡張として提案，実装する．対象言語は OCaml とする．

2　強い型付けを利用したプログラミング環境

2.1　デザインレシピ

デザインレシピとは “How to Design Programs” [1] (以下，HtDP) で提案されているプログラミング手法である．HtDP の 2.5 章 “Designing Programs” には，プログラム開発をする際の手順が載っている．この手順は本学科の関数型言語の授業でほぼそのまま導入されており，また，デザインレシピを使うためのウェブページ [1] も本研究室で開発した．このウェブページでは，ユーザは必要な項目を入力していくだけで OCaml のプログラムのテンプレートを得ることができる．

デザインレシピのうち本論文で注目したのは以下である．

1. これから定義するプログラムの目的を理解し，その入力と出力の関係を “*contract*” (契約) として表す．

2. プログラムのテンプレートを作ることができる

(1) のフェーズはプログラム本体を書く前に行なう作業である．HtDP ではプログラムの入力と出力の関係を “*contract*” と呼ぶ．本来 “*contract*” とは型よりずっと広い概念ではあるが，本論文ではプログラムの型のことを “*contract*” と呼ぶことにする．

(2) は「入力の型から必然的に定まるプログラムの形」のことである．例えば入力にリストが入っていたとき，おそらくこのリストの中身を使うであろうと予想を付けて最初からこのリストをばらす場合分けの構文を入れておくことができる．

OCaml ではプログラムの型をプログラム本体とは切り離して同じソースファイル (`.ml`) に書くことができないため，授業ではプログラムの型をコメントアウト内に書いている．そのためこの *contract* をうまく利用することができず型エラーとなってしまう学生が多々いた．文法的に完全なプログラムを書けた際にはインタラクティブに型エラーの原因を特定し，ユーザに親切なエラーメッセージを出す型デバッガ [2] を使うことができる．

2.2　Agda-mode

Agda では OCaml や Haskell などの関数型言語よりも強い型である依存型を表現できる．そのため型を揃えるのが依存型のない関数型言語よりも困難であることが多い．

Agda-mode はプログラム中に ? (*hole*) を書くことができる．この *hole* の型をコンパイラから得ることにより，ユーザに *hole* やその環境の型を示すだけでなく，*hole* の型や環境の型に合わせた式の自動挿入を行なうことができる．型情報はプログラムが完全に定義される前からユーザから与えられた型注釈によってより具体的な形で推論できる．この型注釈は 2.1 節の *contract* と考えることができる．

3　本システムの実装手法

Agda-mode を使うと型情報を元にして少しずつプログラムを組み立てていくことができる．本論文ではこの特徴に着目し，プログラム中に *hole* を入れることができ，その型によって *hole* の展開や式の挿入ができる Emacs の拡張を OCaml の初学者向けに開発する．本論文で対象とする言語は，本学科の関数型言語で使われる OCaml のサブセットとする．

3.1　システムの外観

本システムは大きく分けて以下の 2 つに分けられる．

1. Emacs 上に現れているインターフェース
   - ユーザから受け取った入力を 2. に渡す
   - 2. の結果に応じてプログラムを変更したりエラーメッセージを出力する

2. OCaml コンパイラのメインのプログラム
   - コンパイラが生成した型情報付きの抽象構文木からプログラム中の任意の式の型を取得
   - 型の定義を見つける

3.2　Typedtree

本開発環境の OCaml のメインプログラムは，Emacs インターフェース側のプログラムからユーザのコマンドを受け取るたびにユーザが書いているソースプログラムを OCaml のコンパイルに渡し，各 `expression` に型情報が付いた抽象構文木 (*Typedtree*) を受け取る．メインプログラムはこの *Typedtree* を登っていくことで *hole* とその環境の型情報を取得する．またその後，型の定義を探索してユーザの送信したコマンドによって定義通りに *hole* の展開や式の挿入を行なう．型の定義は OCaml 内に元々定義されている型の他に，ユーザがプログラム中で定義したものも探索可能である．

[1] http://pllab.is.ocha.ac.jp/~asai/book-mov/recipe.html

4 実装結果

3.1 によって本論文で開発したプログラミング環境では以下の機能が実装できた．

本開発環境では，ユーザが最初にプログラムの型を型注釈で与えることを前提として開発されている．そのため以下に出てくる図では全てプログラムに型注釈が付いている．

```
type person = {name: string; age: int}

let make: string -> int -> person =
  fun s i -> {| }0
```

```
type of hole: person
s : string
i : int
```

図 1. (上) の `hole` の型とその環境の型が出力されている (下).

(a) *hole* の型とその環境の型の表示 図 1 (上) のプログラムは，`string` 型と `int` 型の値を 1 つずつもらって `person` 型の値を返すものであることが，ユーザが書いた型注釈 `string -> int -> person` より分かる．図 1 (下) は *hole* の型とその環境の型を出力したものである．

```
type person = {name: string; age: int}

let make: string -> int -> person =
  fun s i -> {name = {| }0; age = { }1}
```

図 2. 図 1 のプログラムの *hole* を展開する.

(b) *hole* の型に応じた *expression* への展開 図 1 (上) のプログラムで *hole* を返り値の型に展開するコマンドを Emacs 上で送信する．本開発環境は *hole* の型が `person` 型であると特定，定義を探索し，定義の形に展開する (図 2)．*expression* として `tuple`, `record` をサポートした．

```
type person = {name: string; age: int}

let make: string -> int -> person =
  fun s i -> {name = "Yuki"; age = {"25"| }0}
```

```
Cannot Refine:
File "/Users/Yukilshii/lab/expander/test2.ml", line 4, characters 35-39:
Error: This expression has type string but an expression was expected of type
         int
```

図 3. 文法エラーや型エラーが起こると (下) のようなエラーメッセージが表示され，*hole* を外すことができない.

(c) *hole* に入力された値の型と *hole* の型が一致しているかを調べ，一致していたら *hole* を外す 図 3 は図 1 と同じプログラムで，ユーザがレコードの `age` フィールドに `string` 型の値を入れようとした例である．`person` 型の定義により `age` は `int` 型であるのでこれは型エラーとなる．本開発環境はユーザが入力した値をソースコードにそのまま挿入したものを OCaml コンパイラにそのまま渡している．ここで何もエラーが起こらなければ `hole` を外し (図 3 の `name` フィールドはこの例である)，何かエラーが起こった場合はそのエラーメッセージを表示し，`hole` を外せないようになっている．

```
type person = {name: string; age: int}
type tree = Empty
          | Node of tree * person * tree

let search: tree -> person -> bool =
  fun t p -> {t| }0
```

```
type person = {name: string; age: int}
type tree = Empty
          | Node of tree * person * tree

let search: tree -> person -> bool =
  fun t p -> match t with
    | Empty -> {| }0
    | Node (var0, var1, var2) -> { }1
```

図 4. コマンド送信前 (上)，コマンド送信後 (下).

(d) *hole* の環境による場合分けの式の自動挿入 図 4 はユーザが `tree` 型の変数 `t` について場合分けを行う例である．`tree` 型の定義の通りに 2 つのコンストラクタで場合分けをするような `tree` のテンプレートを挿入できている．展開できる型は，授業で使う範囲の `tuple`, `record`, `variant` を対象とした．

(e) その他の式の自動挿入 (d) の `match` 式同様，コマンド 1 つで `if` 式，`let` 式，`fun` 式を挿入する．

5 おわりに

本論文では強い型付けと型注釈を利用して OCaml の開発環境を Emacs の拡張として実装した．今後は本来の目的であった，本学科の関数型言語の授業で使えるようにユーザテストを行い，プログラミング初学者に適した環境に改良をしていきたい．

参考文献

[1] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. *How to Design Programs, Second Edition*. MIT Press, 2014. `http://www.ccs.neu.edu/home/matthias/HtDP2e/`.

[2] Yuki Ishii and Kenichi Asai. Report on a user test and extension of a type debugger for novice programmers. In Proceedings 3rd International Workshop on *Trends in Functional Programming in Education (TFPIE '14)*, Vol. 170 of *Electronic Proceedings in Theoretical Computer Science*, pp. 1–18, 2014.