型デバッガのログの解析とエラーメッセージの改良

石井 柚季, 浅井 健一

お茶の水女子大学

g0920503@is.ocha.ac.jp

asai@is.ocha.ac.jp

概 要 コンパイラの型推論を利用した型デバッガ [7, 9] では，ユーザとの対話によってプログラム中で型エラーの原因となっている構文を特定することが可能である．しかし，初心者プログラマに実際にデバッガを使ってもらうと，特定された構文のどの部分プログラムをどう修正すればよいのかを悩む例が多かった．Marceau は，初心者プログラマにとってエラーメッセージは重要な役割を担っており，従ってどのようなエラーメッセージが初心者プログラマにとってデバッグを行う際に有効であるかを提案している [2, 3]．そこで本論文では，実際に型デバッガが使われた際のエラーログを分析し，それを元にエラーメッセージも含めてデバッガに変更を加えた点について報告する．更に，変更後の型デバッガでも型エラーの原因を特定するのが難しい例について考察する．ユーザテストは，本大学の関数型言語 (OCaml) の授業を履修した学生 40 名を対象とした．

1 はじめに

OCaml や Haskell などの関数型言語の大きな特徴の一つとして，強い型付き言語であることが挙げられる．これは実行前に型エラーを排除できるという利点だけでなく，プログラマがプログラミングをする際の「制約」としての働きも持っている．この「制約」によって，プログラマは「とりあえず型を合わせていってプログラムを動かす」ことを目標に，プログラミングをすることができる．

例えばプログラマが「x に 1 を足した数の x 乗を求めるプログラム」を OCaml で以下のように定義して実行すると，以下のようなエラーメッセージが出力される．

```
fun x -> (x + 1) ^ x

Error: This expression has type int
       but an expression was expected of type string
```

このエラーメッセージは，`^` が `string -> string -> string` 型であるのに対し，`x + 1` が `int` 型となっていることを指摘しているため，プログラマは以下のいずれかを行うことにより「とりあえず動くプログラム」を書くことができる．

1. `^` ではなく `int` 型で累乗を求める関数を適用する
2. `(x + 1)` と `x` を `string_of_int` によって `string` 型にする

ここで最終的に求めたいものは「整数の x 乗（整数）」であるので 1. の方法でプログラムを直せばよいと分かる．

対馬ら [9] はコンパイラの型推論を利用した型デバッガを提案し，実装した．この型デバッガでは，型推論に最汎型木 [5] を用いてアルゴリズミックデバッギング [8] を行うことで，どの構文で型エラーが起きているのかを特定することが可能となった．本論文では，強い型付き言語に慣れていないプログラマにこの型デバッガを使用してもらい，そのエラーログを分析することでより初心者プログラマに使いやすい型デバッガの実装を目指した．

本論文の流れは以下の通りである．まず第 2 節では，型デバッガが型エラーを特定するまでの流れを簡単に述べる．第 3 節では，前述のエラーログを分析，考察する．第 4 節では，初心者プログラマにとってデバッグのしやすいエラーメッセージについて考察し，それらを踏まえて，従来の型デバッガを拡張する．第 6 節では拡張後の型デバッガでテストを行い，改良された点と問題点について述べる．第 7 節で関係研究との比較を行い，最後に第 8 節で今後の課題について述べる．

2 型デバッガの動作

対馬らの型デバッガ [9] がどのように型エラーの原因を特定しているかを，先ほどの例を使って述べる．

2.1 最汎型木

型デバッガが型推論のために採用している最汎型木は Chitil が考案した合成的な型推論 [5] によって得られる型推論木である．先ほどの `fun x -> (x + 1) ^ x` という型エラーのあるプログラムの最汎型木は，以下である．ただし，$\tau^{+} = \texttt{int} \to \texttt{int} \to \texttt{int}$，$\tau^{\wedge} = \texttt{string} \to \texttt{string} \to \texttt{string}$ とする．

$$\dfrac{\dfrac{\boxed{\texttt{\{x : a\}}}^{A} \vdash \texttt{x : a} \quad \boxed{\texttt{\{\}}}^{A} \vdash \texttt{+} : \tau^{+} \quad \boxed{\texttt{\{\}}}^{A} \vdash \texttt{1 : int}}{{}^{(4)}\boxed{\texttt{\{x : int\}}}^{B} \vdash \texttt{(x + 1) : int}} \qquad \texttt{\{\}} \vdash \texttt{\^{}} : \tau^{\wedge}\ {}^{(3)} \quad \texttt{\{x : b\}} \vdash \texttt{x : b}\ {}^{(5)}}{\dfrac{\texttt{\{\}} \vdash \texttt{(x + 1) \^{} x} \cdots \text{型エラー}\ {}^{(2)}}{\texttt{fun x} \to \texttt{(x + 1) \^{} x} \cdots \text{型エラー}\ {}^{(1)}}}$$

通常の型推論木では，型の単一化によって枠 B の {`x : int`} が枠 A へ伝播される．そのため例えばプログラマが {`x : string`} と意図していた場合（つまり，正しいプログラムは `fun x -> (string_of_int (x + 1)) ^ (string_of_int x)` であった場合），どこを直すべきなのかを特定することはできない．しかし，各部分プログラムの型だけを独立して見ていくと，枠 A の部分は {`x : int`} である必要はなく，{`x : a`} としてよい．部分プログラムの型が子ノードの型を合成することでのみ決まる形になっていれば，その後どこでその変数が特定の型になったのかを見ることができる．上の証明木では，枠 A から枠 B へ合成する際に初めて `x` が `int` 型となったことが分かる．よって型エラーの原因は枠 B を含むノードと特定できる．その他の環境や式についても同様に，まず子ノードの型を独立に求め，それらを合成することで親ノードの型を推論している．

2.2 アルゴリズミックデバッギング

型デバッガは最汎型木を用いてアルゴリズミックデバッギングを行うことで型エラーの原因となっている構文を特定している．アルゴリズミックデバッギングは，木構造を持つもののエラー箇所の特定に使われる手法で，元々は Shapiro によって Prolog のプログラムのエラー箇所を特定するために考案された [8]．アルゴリズムは以下のようになる．

1. エラーになっているノードの子ノードにエラーがあるかを見る
2. 全ての子ノードにエラーがなければ，その親ノードがエラーの原因と特定する
3. 子ノードにエラーがあったらそのノードに対してアルゴリズミックデバッギングを行う

エラーがあるかの判定にはユーザの入力を利用する場合が多い．型デバッガの場合には環境と式の型がユーザの意図通りであるかどうかを使っている．

2.3 型エラーの特定

型デバッガは大きく二段階に分けて型エラーの特定を行っている．

1. 全ての子ノード（部分プログラム）には型が付くが，自分自身は型エラーとなっているノード（構文）をアルゴリズミックデバッギングで求める．
2. 全ての部分プログラムはユーザの意図と同じ型を持つが，自分自身はユーザの意図とは異なる型を持つ構文をアルゴリズミックデバッギングで求める．

まず 1. で各部分プログラムの正誤にコンパイラの型推論を用いて構文を特定し，次に 2. ではユーザとの対話によって得られる部分プログラムの型情報を用いて最終的な型エラーの原因を判断する．ここでコンパイラの型推論器を用いることで，型デバッガは 独自の型推論器を実装することなく，OCaml のほぼフルセットをサポートできている [9]．先ほどのプログラム `fun x -> (x + 1) ^ x` の場合，型エラーが起こっている原因には以下の 2 つの可能性が考えられ，`^` についてプログラマが意図している型はそれぞれ以下の括弧内に示す通りである．

a) `^` が累乗を求める演算子であると思っている場合（`int -> int -> int`）

b) `x`, `(x + 1)` に `string_of_int` を適用するのを忘れた場合（`string -> string -> string`）

型デバッガが a), b) それぞれについて型エラーの原因を特定する様子は以下の通りである．最初の 2 ステップはどちらも共通である．

(1) 子ノードに型が付かないので 1 つ上がる

(2) 子ノード全てに型が付いたのでこの先は推論された型がユーザの意図通りかどうかを聞く

a) の場合

(3) 子ノードが意図通りかを聞く「`^` は `string -> string -> string` 型ですか？」
`> no`

(3) 子ノードがないので終了
「`^` は `string -> string -> string` 型です」

b) の場合

(3) 子ノードが意図通りかを聞く「`^` は `string -> string -> string` 型ですか？」
`> yes`

(4) 次の子ノードについて，環境が意図通りかを聞く「`x` は `int` 型ですか？」
`> yes`

(4) 同じ子ノード自体が意図通りかを聞く「`(x + 1)` は `int` 型ですか？」
`> yes`

(5) 最後の子ノードの型 {`x : b`} ⊢ `x : b` は自明なので終了
「この関数呼び出しの `1` 番目の引数で型が合いません」

ここで注目したいのは，`^` の型が a) ではユーザが意図した型とは異なり，b) ではユーザの意図通りであるということである．型デバッガは，ある構文が型エラーの原因と特定するとき，その子ノード全ての型がユーザの意図通りであるがそのノード自身には型が付かない場合を *ill_typed* の型エラー，子ノード全てがユーザの意図通りの型であるが，そのノード自身は（型は付いているが）ユーザの意図通りではない場合を *well_typed* の型エラーとして，それぞれに対応したエラーメッセージを出力している．この例の場合は a) では原因と特定された `^` に型が付いているが意図と違うので *well_typed* の型エラー，b) では特定された関数適用に型が付かないので *ill_typed* の型エラーである．

種類	%	型が付くか
関数適用	27.2	*ill_typed*
match 文	12.1	
コンストラクタの引数	10.2	
if 文	4.1	
再帰関数	3.43	
環境が意図と異なる	17.0	*well_typed*
文法による型エラー	16.3	
特定できず	6.0	
未定義の変数	3.8	

表 1. 構文での分類（ソース数：783）

3　型エラーログの解析

本論文では，対馬らが実装した型デバッガを実際に使ってもらったときのエラーログを解析した．ユーザテストは，本大学の関数型言語の授業を履修した情報科学科の学生 40 名を対象とし，使用した言語は OCaml である．ログは，型デバッガが立ち上がった際のソースプログラムと，ユーザとの対話である．以下の 2 点について解析を行った．

- どの構文が型エラーの原因と特定されたか
- エラーがなくなるまでにどのようなデバッグを行っているか

3.1　構文ごとの解析

どの構文が型エラーの原因と特定されたかで型エラーログを分類した結果は表 1 の通りである．この節では *ill_typed* に分類された型エラーのうち，注目すべき構文について述べ，それらを 5 節で拡張する．

関数適用　圧倒的にエラーの数が多かったのは関数適用の際の引数の型が合わない場合であった．これは，OCaml に限らず強い型付き言語では高階関数を使えることが原因の一つであると考えられる．現在の型デバッガは，関数適用が型エラーの原因だと特定された後，関数に適用する引数を 1 つずつ増やしながら型推論を行い，型エラーが起きたときに増やした引数を原因と特定している．2 節で見たように，*ill_typed* として特定された関数適用の際のエラーメッセージは以下のようになっている．プログラム中の枠はハイライトを示している．

プログラム例：
`fun x ->` `(x + 1) ^ x`[1]
エラーメッセージ：
「この関数呼び出しの 1 番目の引数で型が合いません」（ハイライト 1）

この場合は全ての引数を基底型で考えているのでこれだけでも助けになることはあるだろう．しかし，特に高階関数を使用していた場合，必ずしも型エラーが起きたところの引数が原因とは限らず，厳密にどこが原因となっているのかを特定するのは難しい．エラーメッセージも「i 番目の引数が合いません」とだけが出力されているので，結局関係している全ての箇所を見て自分でデバッグしなければならない．特に目立って多かったのが，引数の数が足りていない場合である．また，先の

ように二項演算子が関数の一つであることに気付かない場合も多く，この場合は「i 番目の引数」が何を指しているのか分かりにくかっただろうと推測される．

match **文**　match 文で目立ったのは，矢印の右側の型が一致していないと思われる場合である．しかし，現在の型デバッガは型エラーの原因として match 文全体を特定するため，その中でどこを直せばよいのかはユーザが自分で考える必要がある．以下は実際にあったエラーログを簡略化したもので，tree_t 型の二分探索木に eki_t 型リストの各要素を挿入する関数である．

```
type eki_t = {kiten : string; shuten : string; kyori : float;}
type tree_t = Empty | Node of tree_t * string * (string * float) list * tree_t
let rec insert_ekikan eki_t_t eki_t =
  match eki_t with                                                    3
    [] -> [] 1
  |{kiten = k; shuten = s; kyori = kyo;} :: rest ->
      match eki_t_t with                                              2
        Empty -> Node (Empty, k, [(s, kyo)],Empty)
      | Node (t1, ekimei, eki_list, t2) ->
         if ekimei < k then
          Node (t1, ekimei, eki_list, insert_ekikan t2 eki_t)
         else if ekimei > k then
          Node (insert_ekikan t1 eki_t, ekimei, eki_list, t2)
            else insert_ekikan eki_t_t rest
```

ユーザは以下のように推論された型を全て意図通りだと回答した．
ハイライト 1 について，{[] : 'a list}
ハイライト 2 について，{k : string, s : string, kyo : float, eki_t_t : tree_t, insert_ekikan : tree_t -> 'a -> tree_t}
エラーメッセージ：
この match 文のどこかがおかしいです（ハイライト 3）

このプログラムはハイライト 1 の [] が原因で外側の match 文の矢印の右側の型が一致していないために型エラーが起こっている．このユーザがこれをデバッグするのに 10 回以上同じ型エラーを出していることを考えると，外側の match 文全体をハイライトしているために（ハイライト 3）下の方の複雑な部分プログラムに目が行き，小さな部分プログラムには気付きにくかったのではないかと思われる．

if **文**　if 文で特に多く見受けられた型エラーは，else 部分がないため then 部分が unit 型でなければならなくなってしまった場合である．大抵のプログラムは else 部分の書き忘れであるが，OCaml に慣れていないユーザはこの型機構を知らない可能性もある．その他，条件部分が bool 型になっていない場合，then 部分と else 部分の型が一致していない場合があった．これら 3 つの場合全て，強い型付き言語ではないプログラミング言語に慣れていると最初のうちは起こりやすい型エラーであると考えられる．しかし，型デバッガのエラーメッセージは全ての場合で「この if 文のどこかがおかしいです」とだけ出るため，ユーザがどこをどのように直せばよいのか悩む例も見受けられた．

	有効な変更	特定された構文内だが無意味な変更	特定された構文外の変更	空白などの無意味な変更
関数適用	36.3	18.2	12.5	33.0
match 文	38.0	10.3	17.2	34.5
コンストラクタ	20.7	17.2	6.9	55.2
if 文	0	50.0	0	50.0
再帰関数	7.7	7.7	23.1	61.5

表 2. プログラムへ加えた変更での分類（%）

3.2 型エラー特定後のユーザの反応

型エラーの原因となっている構文が特定された後，ユーザが型エラーを回避するためにプログラムにどのような変更を加えているかを解析する．プログラムの変更は大きく分けて以下の 4 種類に分類した．この分類方法は [2] を参考にしている．

1. 型エラーを解決するのに有効な変更（型エラー自体はなくならなくてもよい）
2. 型エラーの原因となっている部分プログラムを直しているが型エラーを解決するのには無意味な変更
3. 型エラーの原因とは全く別の箇所を直している変更
4. 前後で変更なし（空白や改行を入れる，を含む）

表 1 で *ill_typed* に分類されたエラーを上記の変更の種類でさらに分類したものが表 2 である．エラーログは，型デバッガが立ち上がるときにのみ取得するようにしたため，連続しているエラーログの最後のソースファイルについては，その後型エラーが直ったのか諦めてしまったのかが分からない．そのため解析の対象としていない．

表 2 から，非常に多くの場合で型エラーの直接の原因となっている部分プログラムとは別の箇所を変更したり，空白を入れるなどの無意味な変更をしていることが分かる．以下，構文ごとに見る．

関数適用　エラーメッセージに「i 番目の引数」という指示が出ているため，その引数を変更したり，引数の順番を逆にしたりして型エラーの回避を試みる例が多かった．しかし，引数が足りないまま他の関数の引数として使用している場合に，引数が不足しているということに気付かない場合も見られた．

match **文**　ハイライトの範囲が広くなるためか，直接の原因とは全く別の箇所を変更する例があった．複雑なデータ構造を使っている場合は match 文が複雑になるため，一度データ構造を簡略化して少しずつ原因に辿り着くユーザや，一度他のソースプログラムを直した後に元のソースプログラムに戻ってプログラムの変更を再開するユーザなど，他の構文よりも時間をかけて丁寧に変更を行っているように見受けられた．

if **文**　2. の変更が多く else 文を書いていない場合に then 文の中の関数適用の引数を変更したり，`unit` 型以外の他の型に変更することが多かった．おそらく OCaml を使い始めたばかりのユーザは「else 文がない場合 then 部分は `unit` 型にする必要がある」ことを知らない場合が多いのではないかと考察される．

3.3 考察

以上の解析から，強い型付き言語に慣れていないプログラマが変更する際に次の 2 つの傾向が見られることが分かる．

- OCaml の型機構を知らないために変更の仕方が分からない
  - if 文の else 部分がない場合
- ハイライトした部分が広いと直接の原因とは異なる箇所を無意味に直してしまう
  - 複雑な match 文

これを元に考察すると，型エラーの直接の原因となっている箇所をハイライトし，なぜ型エラーになるのか（else 部分がないため，など）を出力できるようなエラーメッセージが理想的であろう．「型エラーの直接の原因」としては具体的には，構文の部分プログラムや，変数（環境）までを指示できれば，デバッグはより容易にはなる．また，高階関数を使用して複雑となったプログラムのどこを直すべきなのかを具体的に示すことができれば，より理想的である．以下は実際にエラーログにあったプログラムを例にしている．

```
let rec fold f init lst = match lst with
    [] -> init
  | first :: rest -> f first (fold f init rest)

let length lst = fold (fun l _ _ r -> l + r + 1) 0 lst
```

1 番目の引数である `(fun l _ _ r -> l + r + 1)` を `fold` へ適用させると，全体の型は `'a -> ('b -> int -> int) -> int list -> 'b -> int -> int` となる．ここへ 2 番目の引数 `0` を加えて型推論すると，`('b -> int -> int)` 型と一致しないので型エラーとなる．しかし，このプログラムの場合は 2 番目の引数 `0` が原因なのではなく，直すべきは 1 番目の引数であると考えられる．ここまでをエラーメッセージとして出力できれば，エラーメッセージとしては完璧であろう．

しかし，大学のプログラミングの授業で使用するデバッガとして，そこまでの出力が適切であるかは議論の余地がある．次節では，Marceau の分析 [2, 3] を元に初心者プログラマ向けのエラーメッセージとして適切な設計を考察する．

4 初心者プログラマ向けのエラーメッセージ

Marceau らは [2] で，Racket を使った初心者プログラマ向けのエラーメッセージを分析，考察している．特に注目したい分析は以下である．

- 英語が母国語の学生たちに英語で書かれたエラーメッセージを見せたが，一つ一つの単語の意味は分かっても全体として何を言われているのか分からない場合があった
- プログラミングや関数型言語に特有の単語は, 学生のそれまでのプログラミング環境によっては知らない場合もあった
- ”This expression” などの指示語が何を指しているのか分からない場合があった

また Marceau ら [3] は教育的な観点から，エラーメッセージを設計する際の注意点を 2 つ挙げている：

- エラーメッセージはエラーの解決法を示してはならない．これは，例えよくある（閉じ括弧がないなどの）エラーであっても，示した解決法が本当に全ての場合をカバーしているとは限らず，余計に誤ったプログラムに導いてしまう可能性があるからである．
- エラーメッセージはプログラマに誤ったデバッグをさせてはならない．これは例えば，ハイライトによってプログラマに「ハイライト内のみを直せばよい」と思わせてはならないということである．

本論文でも現段階で型デバッガのユーザ対象としているのは情報科学科の学生であるため，上記の2点とおおよそ同じ観点でエラーメッセージの設計をする．さらに，OCaml の型機構を理解し，型を揃えることに慣れてもらうことを設計する際の目的の一つに加える．そのために，エラーを起こしている構文だけでなく，その中のどの部分プログラムが原因となっているのかを特定する必要がある．これにより，エラーメッセージには特定された部分プログラムの型，それ以外の関連していると考えられる部分プログラムの型を出力し，なぜ型エラーが起こっているのかを示せるようになる．例えば，以下のようなプログラムの一部を書いたプログラマがいたとする．

```
...  if p < q then p + 1 else "q"
```

このプログラマが OCaml では if 文の then 部分と else 部分の型を一致させなければならないことを知らなかった場合，エラーメッセージとして「if 文が型エラーを起こしている」とだけを示されてもすぐにデバッグはできないかもしれない．さらに「then 部分が `int` 型で else 部分が `string` 型」であることのみを示されても，（型エラーにならない言語もあるのに）これでなぜ型エラーが起こるのかは慣れていないうちは分からない可能性がある．そこで「OCaml では if 文の then 部分と else 部分の型は一致させなければならない」ということも一緒に出力し，OCaml の型機構を理解する機会を作るようにする．

この問題の最も重要で効率的な解決法は，強い型付き言語に慣れていないプログラマに，型の概念をしっかりと身に付けてもらうことである．これについては，8 節で議論する．

5　型デバッガの拡張

本節では，型デバッガを初心者プログラマ向けに拡張した部分を見ていく．2 節で見たように，型エラーのある構文を特定するためには最汎型木が必要であった．*ill_typed* の型エラーとして特定された場合にどの部分プログラムが原因であるかを特定するには，特定された構文に OCaml の型機構に沿った型を追加していくことで可能になる．例えば if 文の条件部は `bool` 型，then 部分と else 部分は同じ型である必要がある，などの制約を部分プログラムが満たしているか判定し，その結果をエラーメッセージとして出力する．これはつまり，最汎型となっている構文に少しずつ OCaml の型機構を制約として追加していくことと同じである．

拡張は，*ill_typed* の型エラーとして特定された if 文，match 文，関数適用でどの部分プログラムが型エラーの原因となっているのかを特定できるようにした．このとき，従来の型デバッガと同様に型推論には OCaml のコンパイラの型推論器を用い，OCaml のように大きな言語に対してもフルセットをサポートできるという特徴を踏襲している．以下，拡張したそれぞれの構文について見ていく．拡張後の型デバッガはまだ授業でユーザテストを行っていないため，各プログラム例で型デバッガへの回答は拡張前と同じと仮定し，型エラーの原因となっている部分プログラムをどのくらい特定できているかを検討した．

5.1 if 文

5.1.1 条件部

OCaml の条件部は bool 型となっている必要がある．そこで，型エラーを起こしている if 文の条件部に bool 型とアノテーションを付けて，条件部のみを再度コンパイラの型推論にかけてみる．もしこのとき型が付かなかったら条件部が原因で if 文に型が付かないと判断し，エラーメッセージを出力する．ハイライトは，if 文全体ではなく条件部のみをハイライトする．以下は実際にあった，ユーザが条件部を float 型でよいと意図していた場合のプログラムの一部である．

プログラム例：
```
...  if (try kekka_kyori with Not_found -> infinity) [1] then ...
```

エラーメッセージ：(ユーザは，推論された型を全て意図通りであると回答した場合)
条件部分の型は float ですが bool 型でなくてはなりません（ハイライト 1）

このように条件部の実際の型と，bool 型である必要があることを一緒に出力し，デバッグの解決法と OCaml の型機構への理解を同時に促せるようにした．

5.1.2 else 文がない場合

OCaml は if 文の else 以下を省くことができるが，その場合 then 部分は必ず unit 型となっている必要がある．そこで条件部と同じように then 部分に unit 型とアノテーションを付けて型推論を行い，型が付かなかったら then 部分が原因であると判断する．エラーメッセージは「then 部分が unit 型である必要があること」の他に「else 部分を書き忘れている可能性があること」を出力し，OCaml の型機構に慣れていないプログラマにデバッグの方針の一つを提案している．

プログラム例：
```
...  if (a + 1) < c then a + 1 [1]
```

エラーメッセージ：(ユーザが a，c を int 型であると回答した場合)
then 部分が int 型になっていますが else 部分がないため unit 型でなくてはなりません
（もしくは else 部分を書き忘れていませんか？）（ハイライト 1）

5.1.3 else 文がある場合

if 文に else 部分がある場合，then 部分と else 部分の型は一致していなければならない．そこで，then 部分，else 部分に同じ型とアノテーションを付けるのだが，本論文ではこれらを同じリストに入れている．このリストをそのまま型推論し，リストに型が付かなかったら then 部分と else 部分は同じ型になり得ないと判断し，エラーメッセージを出力する．このとき，最汎型木で推論された then 部分，else 部分の型を一緒に出力している．

プログラム例：
```
...  if (a + 1) < c then a + 1 else print_int c [1]
```

エラーメッセージ：(ユーザが a，c を int 型であると回答した場合)
then 部分が int 型で else 部分が unit 型になっていますが これらは同じ型でなくてはなりません（ハイライト 1）

5.2 match 文

OCaml の match 文は，*(exp, (pattern, expression) list)* という構造をしている．ここで match される式を *exp*，match した後の矢印の左側を *pattern*，右側を *expression* としている．OCaml の macth 文で必要とされる型機構は以下の 3 つである．

1. 全ての *pattern* が同じ型であること
2. *exp* と *pattern* が同じ型であること
3. 全ての *expression* が同じ型であること

これらを用いて match 文のどの部分プログラムが型エラーの原因となっているのかを特定するために，*(pattern, expression) list* の要素を後ろから削っていき，型エラーを引き起こす要素を探している．具体的なアルゴリズムは以下の通りであるが，最初に *pattern* について調べるために，*expression* にダミーの値を入れて型を揃えている．

1. 全ての *expression* を型の揃うダミーの値に置き換えた状態で型推論にかける．型エラーが起こらなかったら 5. へ．
2. *(pattern, expression) list* から一つ要素を取り除いた状態で型推論にかける．
3. 型エラーが起こらなくなるまで 2. を繰り返す．
4. 型エラーが起こらなくなる直前まで *(pattern, expression) list* に入っていた *pattern* が型エラーの原因と特定する．もし最後の要素まで型エラーが起こっている場合は match される *exp* と *pattern* の型が合わないと判断する．
5. 全ての *expression* を元に戻し，型エラーがなくなるまで 2. を繰り返す．
6. 型エラーが起こらなくなる直前まで *(pattern, expression) list* に入っていた *expression* が型エラーの原因と特定する．

エラーメッセージには，特定された部分プログラムをハイライトした上で，「この部分プログラムの型とそれ以前までの *pattern / expression* の型が一致しない」もしくは「*exp* と *pattern* の型が一致しない」ことを出力する．特定された部分プログラム以前の *pattern / expression* の型を表示するのは，必ずしも特定された部分プログラムがユーザが直すべき箇所とは限らないからである．

3.1.2 節の match 文のプログラムと同じものを拡張後の型デバッガで試すと，以下のようなハイライトとエラーメッセージが出力される．

ハイライト部分が `ekikan_tree_t` 型でそれ以前までの矢印の右側が `'a list` 型になっていますが、これらは同じ型でなくてはなりません（ハイライト 2）

ここでも「同じ型である必要がある」ことを示すことで，OCaml の型機構を理解し型を合わせながらプログラムを書くことに慣れてもらえるようにした．

5.3 関数適用

関数適用の際は，関数に引数をまず 1 つ渡し，全体に型が付いたら引数を 1 つ増やし…を型エラーが起こるまで繰り返すことで型エラーの原因を特定していた．しかし関数型言語では高階関数を使えるので，関数適用のどの部分プログラムが型エラーの原因となっているのかを厳密に特定するのは困難である．そこで，関数の型，各引数の型，型エラーとなる直前まで引数を渡したときの式の型を出力するようにエラーメッセージを変更した．

最初の `fun x -> (x + 1) ^ x` というプログラム例を拡張後の型デバッガで試すと，*ill_typed* として特定された場合のエラーメッセージは以下のようになる．

部分プログラム	%
条件部	3.1
then 部が `unit` 型でない	78.1
then - else 部	9.4
failed	9.4

表 3. if 文

部分プログラム	%
pattern の型	0
exp と *pattern* の型	4.9
expression の型	80.4
failed	14.7

表 4. match 文

引数の型が合わない	98.4
failed	1.6

表 5. 関数適用 (%)

この関数呼び出しの 1 番目の引数で型が合いません。
関数の型、引数の型、0 番目までの引数を関数に渡した式の型は以下の通りです。
関数部分： `string -> string -> string`
1 番目の引数の型： `int`
2 番目の引数の型： `'a`
0 番目までの引数を関数に渡した式の型： `string -> string -> string`

この場合は関数の型と引数の型だけでも，なぜ 1 番目の引数で型エラーが起こったか分かるであろう．しかし，型エラーとなる直前まで引数を渡した式の型と引数を見比べることで，次の引数の型として何が必要で，実際にはどの型の引数を渡そうとしているのかをユーザ自身が見つける際の更なる一助になると期待している．3.3 節の高階関数 `fold` を使った例ではこのようなエラーメッセージが出る．

この関数呼び出しの 2 番目の引数で型が合いません
関数の型、引数の型、1 番目までの引数を関数に渡した式の型は以下の通りです
関数部分：`('a -> 'b -> 'b) -> 'b -> 'a list -> 'b`
1 番目の引数の型： `int -> 'c -> 'd -> int -> int`
2 番目の引数の型： `int`
3 番目の引数の型： `'e`
1 番目までの引数を関数に渡した式の型：`('f -> int -> int) -> int list -> 'f -> int -> int`

関数の型と引数の型だけで，型に慣れていないユーザになぜ 2 番目の引数が問題なのかを答えさせるのは難しいであろう．型エラーが起こる直前の型を示すことで，次に期待されている型と i 番目（この例では 2 番目）の引数を比べてもらい，どちらが原因なのかを判断できるようにという狙いがある．

6 拡張後の型デバッガの解析と考察

本節では，拡張後の型デバッガを使って if 文，match 文のどの部分プログラムが原因だと特定されたかを分類し，その後関数適用も含めて，うまく動作しなかった場合や，メッセージとして今ひとつであった場合について考察する．まず，if 文と match 文の分類結果は表 3，表 4 である．表 3 〜 表 5 中で，”failed” という項目が，型デバッガがうまく動作しなかった場合である．

それぞれの表から，多くの例で拡張後の型デバッガは型エラーの原因をより詳しく説明できていたことが分かる．以降，”failed” となってしまったプログラムの解析と，エラーメッセージが今ひ

とつ分かりにくい例を考察する．

6.1 環境の衝突

if 文，match 文で型デバッガがうまく動作しなかった例の多くが，環境が衝突している場合である．つまり，各部分プログラムは型が付き，かつ，ユーザの意図通りではあるが，部分プログラム間で同じ変数に異なる型を与えている場合である．

```
fun p -> fun q -> if p and (q = 1) then p else q
```

というプログラムを例に考えてみる．型エラーの原因が if 文に特定された後，型デバッガは以下の処理を行う．

1. 条件部 `p and (q = 1)` に `bool` 型とアノテーションを付けて条件部のみを型推論する

   - {`p : bool, q : int`} のとき成立する

2. then 部分 `p` と else 部分 `q` を同じリストに入れたままリストごと型推論する

   - {`p : 'a, q : 'a`} で成立する

このプログラムは各部分プログラムが，そこだけ見ると OCaml の型制約を満たしている．しかし，1. の環境 {`p : bool, q : int`} と 2. の環境 {`p : 'a, q : 'a`} を同時に満たすような環境を作ることはできない．if 文でも match 文でも，failed となったプログラムの全てがこのような環境の衝突を起こしていた．match 文の場合には各 *expression* の間で環境が衝突している場合が半分以上あり，その他では match される *exp*，*pattern*，*expression* の間で環境の衝突が見られた．

各部分プログラムの環境を単一化するようなプログラムを自分で書けば，このような型エラーに対しても対応はできる．しかしそれは，型デバッガの「OCaml の型推論器を利用する」という方針に反することになり，従って実装には多くの手間を要する．OCaml の型推論器を利用することで「実装にかかる手間が省ける」ことと「OCaml の型推論器との推論の食い違いを防ぐ」という型デバッガのメリットを活かし続けるために，環境が衝突している際にどのように対応するかは今後検討する必要がある．

6.2 型変数があるとき

関数適用が特定された際には，関数，全ての引数，型エラーとなる直前まで引数を渡した式の型を出力したことで，ユーザが原因を探しやすくなったと期待している．しかし，以下のようなプログラムの場合はそれがうまくいかない．

```
type tree_t = Empty | Node of tree_t * char * int * tree_t
let rec search tree name = match tree with
    Empty -> Empty
  | Node (t1, st, n, t2) -> if (st = name) then n
                            else search search t2 name
```

このプログラムは枠の部分の関数適用が型エラーを起こしている．ここで型デバッガを立ち上げると，ユーザとの対話が一度もないまま以下のエラーメッセージが出力される．

```
ハイライトされた部分に特定されました
この関数呼び出しの 1 番目の引数で型が合いません
関数の型、引数の型、0 番目までの引数を関数に渡した式の型は以下の通りです
```

関数部分： 'c
1 番目の引数の型： 'd
2 番目の引数の型： 'e
3 番目の引数の型： 'f
0 番目までの引数を関数に渡した式の型： 'g

全ての型が型変数となってしまっているのでエラーメッセージを読んでも原因を特定するのは非常に難しいであろう．ユーザとの対話がないのは，rec とあるので search の body 部分で search は単相となり，ハイライト部分の search search のところで型が付かず，さらに部分プログラムの search は変数であるため自明に型が付くからである．また，全てが型変数になってしまうのは，部分プログラムが全て変数であるためである．

また関数適用ではなく if 文でも型変数によりメッセージが分かりにくい例がある．

```
(* take_less : int -> int list -> int list *)
let rec take_less n lst = match lst with
    [] -> []
  | first :: rest ->
          if n > first then first :: take_less n rest else take_less rest
```

エラーメッセージ：(ユーザが推論された型を全て意図通りだと答えた場合)
then 部分が 'a list 型で else 部分が 'b 型になっていますが，これらは同じ型でなくてはなりません

このプログラムで直すべき箇所は，else 部分の関数適用の引数を 1 つ増やすことである．しかし，このエラーメッセージからその解決法へ至るのは少し難しいのではないかと推測される．これは，take_less が rec によって単相であること，rest が 'a list 型と 'a 型として使われていて型が付かないことが原因である．これらの例のように，型変数が多く出てきてしまうような場合にどのように対処するかは現段階ではまだ検討できていない．

7 関連研究

初心者プログラマのデバッグ時の挙動についてのユーザテストは様々な言語を使って数多く行われている．関数型言語を使ったものでは Marceau らの Racket を使ったユーザテスト報告がある [2, 3]．ここでは Racket を使ったプログラミング言語の授業においてプログラミング中にエラーが起こったときのユーザの反応を動画で保存し，さらにエラーメッセージについてのアンケートを行い，各エラーメッセージの有効性についての分析，考察を行っている．詳細は本論文の 4 節で述べた通りである．エラーメッセージではなく，Java を使ったデバッグの手段についてその有効性を調べた研究 [4] では，デバッグの手段を 12 種類に分類し（例：標準出力を使うなどして状態の変化を見る，JavaDoc やデバッガを使う，コメントアウトによって問題を独立させる，など），それぞれどのような使い方をすると有効なデバッグが可能となるかをユーザテストを元に検証している．本論文でのデバッグの手段は型デバッガであり，手段そのものよりも型デバッガが出すエラーメッセージへの反応を見ている点で本論文とは目的が少し異なる．初心者プログラマを対象とした Pascal のバグ解析を詳細に行った研究もあり [6]，バグを構文ごとに分類した他に，ユーザの意図した解決法（"plan"）と実装との相違による分類の 2 つの分類方法を使っている．plan と実装の相違での分類は，バグは「plan は正しいが実装を間違えている」場合と「plan も実装も間違えている」場合の

2 種類にさらに分類される．本論文でもユーザの意図を元に型エラーの原因を特定しているため，ユーザの意図を考慮したバグの分類はとても興味深い．

8 今後の課題

今回拡張したのは *ill_typed* で特定された構文の一部であるが，他の構文についても問題があることが分かっているため，今後の課題としてまずそれらの構文についてのエラーメッセージの改良がある．その他，以下のような課題を検討している．

8.1 高階関数の使用制限

本大学の関数型言語の授業では，最初の何回分かは高階関数を使わない．おそらく多くの関数型言語の書籍やウェブサイト，授業なども同様であろう．そこで，この一番最初の期間では高階関数を敢えて使えなくするように OCaml に制限を加えた言語を実装することを考えている．Racket は既に 5 段階に言語レベルを分けていて，初心者向けの言語レベル（“beginning student”）では高階関数の使用を禁止している．これにより，より単純な型をエラーメッセージとして出力することができ，さらに高階関数を使用した場合に他のエラーメッセージではなく直接「引数が不足していないか」と示す事もできる．さらに，この段階では else 部分のない if 文を書くことも禁止する．これは，`unit` 型を扱うプログラムが最初の方では出てこないからである．この期間で関数型言語を初めて扱うユーザには「型を揃える」ということに慣れてもらい，その後の高階関数や再帰，副作用の入った複雑なプログラムを書くときの助けになればと期待している．

8.2 *well_typed* に分類されたもののエラーメッセージの改良

8.2.1 構文ごとに改良する

現在の型デバッガは，*well_typed* で特定された構文の部分プログラムのうち，推論された型に対してユーザが `no` と答えた自由変数や式の推論された型を「変数 `p` は `'a` 型になっています」という書式で出力している．これは，どの構文が特定されても同じエラーメッセージである．しかし，表 1 ではこのタイプの（環境が意図通りでない）エラーが非常に多い．そこで，OCaml の型制約の理解を深めてもらう機会を増やすためにはここでも *ill_typed* と同様，構文ごとにエラーメッセージを変更する必要がある．

型デバッガは，推論された型に対するユーザの回答をリストにして保存している．*well_typed* に分類された場合のエラーメッセージは，このリストを利用することで可能になると考える．例えば else 文のない if 文が *well_typed* の型エラーとして特定された場合，このリストから then 部分の型に対するユーザの反応を探し，ユーザが then 部分を `unit` 型ではないと回答していた場合に「else 部分がないため，then 部分は `unit` 型になっています」などのメッセージを出す．

8.2.2 文法による型エラー

表 1 の「文法による型エラー」は，「環境がユーザの意図と異なる」に分類されたもののうち，以下が原因で型エラーになっているものを特別に集めたものである．

- リストをタプルのリストにしてしまう（例： `[1; 2]` → `[1, 2]`）
- タプルを `float` 型にしてしまう（例： `(1,2)` → `(1.2)`）
- `type tree_t = Empty | Node of tree_t * 'a * tree_t`
  を宣言後に `Node` を書かずに `tree_t * 'a * tree_t` 型にしてしまう
  （例：木の要素全てを 2 倍にする関数 `double_tree` で `Node (double_tree, 2 * n, double_tree)` → `(double_tree, 2 * n, double_tree)`）

- `ref` 型の中身を参照する際に `!` を書いていない
  （例：`counter := !counter + 1` → `counter := counter + 1`）
- 例外を起こす際に `raise` を書いていない
  （例：`[] -> raise Not_found` → `[] -> Not_found`）

これらのエラーは数も非常に多いが，OCaml に慣れていないプログラマにとってはデバッグの仕方が分からず，なかなか型エラーが直らないという傾向があった．そこで Lerner らの手法 [1] を参考に，型エラーとなっている式を，文法が似ていて誤りやすい式に変更し，型が付いたらその式と間違えている可能性を示すことを検討中である．

8.3 拡張後の型デバッガのユーザテスト

まずは拡張した型デバッガを来期の関数型言語の授業で使ってもらい，どのくらい有効であるかを見る．これによってさらに詳細な解析が可能になると考えられるが，そのエラーログの多さから，今回の解析は非常に手間がかかった．そのため，解析しやすいログの取り方を検討している．現在はユーザとの対話とソースプログラムを日付と時間で分類しているが，これを「どのエラーメッセージが出力されたか」によって自動で分類できるようにする．これは，エラーメッセージに固有の番号を振っておき，エラーメッセージが出力される際にその番号も一緒に出力するようにする（ただしユーザが見ている画面の範囲外に出力する）ことで可能になると考えている．

謝辞 多くの有益なコメントを下さった査読者の皆さまに深く感謝致します．

参考文献

[1] Benjamin Lerner, Dan Grossman, and Craig Chambers. Seminal: Searching For ML Type-Error Messages. In *ML '06 Proceedings of the 2006 workshop on ML*, pp. 63–73, 2006.

[2] Guillaume Marceau, Kathi Fisler, and Shriram Krishnamurthi. Measuring the Effectiveness of Error Messages Designed for Novice Programmers. In *SIGCSE '11 Proceedings of the 42nd ACM technical symposium on Computer science education*, pp. 499–504, 2011.

[3] Guillaume Marceau, Kathi Fisler, and Shriram Krishnamurthi. Mind Your Language: On Novices' Interactions with Error Messages. In *ONWARD '11 Proceedings of the 10th SIGPLAN symposium on New ideas, new paradigms, and reflections on programming and software*, pp. 3–18, 2011.

[4] Laurie Murphy and et al. Debugging: The Good, the Bad, and the Quirky – a Qualitative Analysis of Novices' Strategies. In *SIGCSE '08 Proceedings of the 39th SIGCSE technical symposium on Computer science education*, pp. 163–167, 2008.

[5] Chitil O. Compositional Explanation of Types and Algorithmic Debugging of Type Errors. In *ICFP '01 Proceedings of 6th ACM SIGPLAN International Conference on Functional Programming*, pp. 193–204, 2001.

[6] James C. Spohrer and Elliot Soloway. Novice Mistakes: Are The Folk Wisdoms Correct? *Communications of the ACM*, Vol. 29, pp. 624–632, 1986.

[7] K. Tsushima and K. Asai. An Embedded Type Debugger. In *Implementation and Application of Functional Languages, Lecture Notes in Computer Science 8241 (IFL' 12)*, pp. 190–206, 2012.

[8] Shapiro E. Y. Algorithmic Program Debugging. In *MIT Press*, 1983.

[9] 対馬かなえ, 浅井健一. コンパイラの型推論を利用した型デバッグ手法の提案. In *The 13th JSSST Workshop on Programming and Programming Languages*, 2012.