型デバッガのユーザテスト結果と拡張

石井柚季　(指導教官: 浅井健一)

1　はじめに

コンパイラの型推論を利用した型デバッガ [3, 2] 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．本研究では，本大学の関数型言語の授業で OCaml を初めて使う学生 40 名に型デバッガのユーザテストを行った際の型エラーログを解析し，それを元に if 文，match 文，関数適用の構文について型デバッガの拡張を行った．

2　型デバッガ

型デバッガ [3, 2] は，プログラム中の型エラーが起きている部分について，意図している型をユーザに質問することで型エラーの原因となっている構文を特定する．例えば以下の型エラーのある OCaml のプログラムがあるとする（プログラム中の四角枠はデバッグ時のハイライトを示している）．

```
fun x -> [ (x + 1)3 ^1 x2 ]4
```

ここでユーザが `(x + 1)`, `x` に `string_of_int` を適用するのを忘れていた場合は以下のような対話になる（括弧内の数字は上のプログラムのハイライトと対応している）．

1. `^` は `string -> string -> string` 型ですか？(1) `> yes`
2. `x` は `int` 型ですか？(2) `> yes`
3. `(x + 1)` は `int` 型ですか？(3) `> yes`
4. この関数呼び出しの `1` 番目の引数で型が合いません (4)

このように型デバッガは，ユーザが意図している型を用いることでどの式を修正すればよいのかを特定している．

3　型エラー特定後のユーザの反応

本研究では，2012 年度の関数型言語の授業を履修した学生 40 名を対象に型デバッガのユーザテストを行った際の型エラーログを解析した．型エラーの原因が特定された後，ユーザがプログラムをどのように変更するかを [1] を参考に以下のように分類した（分類結果：表 1）．

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箇所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）

	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 1: プログラムへ加えた変更での分類（%，ソース数:783）

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，引数が足りていないことが型エラーの原因である場合は，そのことに気付かないエラーログも見られた．

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．

4　型デバッガの拡張

今までは文単位で特定し，エラーメッセージを出力していたが，前節で考察を行った構文について，構文のどこ（どの部分プログラム）に型エラーの原因があるかを特定できるよう型デバッガを拡張した．match 文と if 文では各部分プログラムが OCaml の型機構に沿った型をもっているかを型注釈を付けて確認することで構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かを特定できるようにした．また関数適用は，それまで「n 番目の引数がおかしいです」とだけ出ていたエラーメッセージを，部分プログラムの型も出力するように変更した．表 2〜表 4 は，それぞれの構文でどの部分プログラムが型エラーの原因だと特定されたかを分類したものである．これらの表より，非常に多くの場合で拡張後の型デバッガがうまく型エラーの原因となっている部分プログラムを特定できていることが分かる．failed となってしまったプログラムについては，次節で考察を行う．以下，構文単位から部分プログラム単位での型エラーの特定ができるように拡張した if 文，match 文ついてエラーメッセージがどのように変更されたかを見る（いずれの場合も，ユー

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 2: 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 3: match 文

引数の型が合わない	98.4
failed	1.6

表 4: 関数適用（%）

ザは推論された型全てについて意図通りであると回答したことを想定している）.

4.1 if 文

以下のプログラムは，if 文の else 以下がないために型エラーが起こっている．これについて，拡張前後でエラーメッセージは以下のように変更されている．

プログラム例：

```
...  if (a + 1) < c then a + 1 [2]  [1]
```

拡張前：
この if 文のどこかがおかしいです（1）
拡張後：
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？）（2）

4.2 match 文

match 文では全ての *expression* の型が同じでなければならない．例として以下のプログラムを見ると，拡張後では部分プログラム単位での特定ができている．

プログラム例：

```
let rec f lst = match lst with                    [1]
    [] -> []
  | fst ::  rst -> fst + (f rst) [2]
```

拡張前：
この match 文のどこかがおかしいです（1）
拡張後：
ハイライト部分が int 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（2）

5 型デバッガがうまく動作しない原因

5.1 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エラーを起こさず，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．解決策として独自の型推論器を作ることが挙げられるが，OCaml コンパイラの型推論結果との不一致が起こる可能性や，実装の困難さを考えるとできるだけこれは避けたい．

5.2 型変数がある場合

今回拡張した構文では，各部分プログラムの型を出力しエラーメッセージの理解の向上を図っている．しかし，出力された型全てが型変数となってしまう型エラーもあった．この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．

6 まとめと今後の課題

本研究では OCaml を初めて使うプログラマにデバッグ時に型デバッガを使用してもらい，その型エラーログの解析と考察を元に型デバッガを拡張した．今後さらに型デバッガを効果的に利用できるように，以下のような課題を検討している．

6.1 高階関数の使用制限

本大学の関数型言語の授業では，最初の何回かは高階関数を扱わない．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．例えば Racket は既に 5 段階に言語レベルを分けていている．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る．さらに，この段階では unit 型を扱うプログラムが出てこないため else 部分のない if 文を書くことも禁止する．関数型言語を初めて扱うユーザにはこの期間で「型を揃える」ということに慣れてもらい，その後高階関数や再帰，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．

6.2 型エラースライスの表示

型エラーのあるプログラムうち，型エラーに関係している箇所のみを抽出したプログラムを型エラースライスと呼ぶ [4]．型エラーが起こった際にこの型エラースライスをグラフの形で表示し，各ノード（式または部分プログラム）に推論された環境と，その下でのそのノードの型を見られるようにし，更にユーザが意図している型を追加できるようにする．これによりユーザへの質問なしでユーザの意図を把握でき，エラーメッセージが分かりにくいという問題も解決できると考えている．

参考文献

[1] Guillaume Marceau, Kathi Fisler, and Shriram Krishnamurthi. Measuring the Effectiveness of Error Messages Designed for Novice Programmers. In *SIGCSE '11*, pp. 499–504, 2011.

[2] K. Tsushima and K. Asai. An Embedded Type Debugger. In *IFL '12, LNCS 8241*, pp. 190–206, 2012.

[3] 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 *PPL '12*, 2012. 15 pages.

[4] 対馬かなえ, 浅井健一. コンパイラの型推論を利用した型スライス作成手法の提案. In *PPL '13*, 2013. 15 pages.