型エラースライシングを利用した型エラーデバッガに関する実装と考察

脇川奈穂　(指導教員：浅井健一)

1　はじめに

本研究では、OCaml を対象とする型エラーデバッガに型エラースライシングを実装し、それの効果や有用性について考察を行う。

OCaml などの強い型付き関数型言語では、型推論によって多くの実行時エラーを取り除く。しかし、コンパイラが返すエラーメッセージと型エラーの原因は相違する場合が多く、その際にプログラマは型エラーの原因を自力で探さねばならない。対馬ら [3] の型エラーデバッガは、型エラーの原因を探す上で大いに役立つが、挙動に無駄があることが問題点であった。

2　型エラーとその原因

具体例として、OCaml で書かれた下記のプログラムの実行結果を考える。

```
let sum x y = x + y in sum 1 2.0
Error: This expression has type float
but an expression was expected of type
int
```

このプログラムを実行すると、プログラム中の下線部が引かれた部分で型の衝突が起こる。型エラーは、二つの式の型の衝突と、それらを同一でなければならないと強制する箇所が存在することによって発生する。型エラーは型の衝突によって発生し、エラーメッセージは下線部で `int` 型が期待されたが実際は `float` 型であると指摘している。

この型エラーを修正する方法は、プログラマの思い描くプログラムによって無数に考えられる。プログラマの考える原因が `2.0` にある場合、ユーザはエラーメッセージから型エラーの原因を理解し、プログラムを正しく変更できる。しかし、それ以外の場合、ユーザは型エラーの原因を自ら探さなければならない。

3　型エラーデバッガ

Chitil[1] は合成的な型推論によって対話的な型エラーデバッガを可能とし、対馬らはそれにコンパイラの推論器を用いる手法を提案した。前述のプログラムを対馬らの型エラーデバッガ上で実行すると下記のようなメッセージが表示される。

```
let sum x y = x + y in sum 1 2.0;;
ハイライトされた部分の型を int -> int -> int
だと意図していますか?
(y/n/q) >
```

この質問では、ユーザが関数 `sum` の型を `int -> int -> int` であると意図しているかどうかを尋ねている。ユーザはそのように意図していれば y(yes)、そうでなければ n(no) と回答すればよい。デバッグは型の衝突があった箇所に対する質問から開始し、以降はユーザの回答に応じて質問される箇所や原因と特定される箇所が変化する。また、これらの該当箇所を求める際に、アルゴリズミックデバッギング [2] を利用している。

このデバッガの問題点は、原因になり得ない箇所に対する質問が行われることと、レコードやコンストラクタのような内部に複数の型を持つ式を含む場合に質問回数が多くなりがちなことである。上の例では、`1` や `x` は型エラーと関係ないため、これらに対する質問は無駄であるといえる。レコードやコンストラクタに対する質問は、式が持つ全ての要素に対して行われるため、要素数が多いほど質問回数も多くなる。

4　型エラースライス

対馬らの型エラーデバッガの問題点は、型エラースライスの導入によって解決される。型エラースライスとは、型エラーに関係する部分のみ抜き出したプログラムである。前述のプログラムの型エラースライスは以下の通りである。

```
let sum x y = □ + y in sum □ 2.0
```

型エラーに関係ない部分は、抽象化という処理によって□に置き換えられる。□は「型が何でも良い」多相型の式を意味し、多相型はユーザに意図を尋ねる必要がないため、結果的にその部分に対して質問されなくなる。上の例では、型の衝突が `2.0` と `+` で発生したため、関係のない `x` と `1` が□に置き換えられている。

5　提案手法

式中の型エラーと関係ない部分を抽象化することによって、質問回数の減少によるデバッグの効率化が期待できる。質問が必要か否かは、型エラースライスで抽象化されているか否かと同義である。そこで、本研究では対馬ら [4] の型エラースライサ（型エラースライスを求めるプログラム）の実装および構文の拡張をし、型エラーと関係のない箇所に対する質問を省略する。

型エラースライサは、アルゴリズムデバッギングを元にした以下のアルゴリズムに従って型エラースライスを求める。部分的に抽象化とは、式中に部分式の個数分、1 つだけを抽象化したような式の集合を求めることである。

1. ある式を部分的に抽象化する。
2. 1. で求めた集合に型エラーとなる式があった場合は該当する要素の式、なかった場合は各部分式に対して、このアルゴリズムを再帰する。
3. 複数行に渡るプログラムの場合、手前の行と着目している行を合わせてひとつの式とみなし、このアルゴリズムを再帰する。

対馬らの型エラースライサに元からある構文は、定数、変数、let 文、fun 文、関数適用である。これらのうち、定数と変数は部分式を持たず、部分的に抽象化した際には 唯一の要素として □ が得られる。本研究では新たに追加する構文として、レコード、コンストラクタ、パターンマッチを追加する。

5.1　レコード

例:`type t1 = {a: int; b: int; c: string; }`

フィールドごとに決められた型の要素を持つ。各要素を部分的に抽象化する。

5.2 コンストラクタ

例：`type t2 = X of int * int * string`

要素を持つ場合は各要素を部分的に抽象化する。要素を持たない場合は定数として扱う。

5.3 パターンマッチ

例：`match m with p1 -> n1 | p2 -> n2`

パターンマッチする式 `m`、結果の各式 `n1`、`n2` を部分的に抽象化する。`p1`, `p2` はパターンという式とは異なるものとして扱う。

6 型エラースライサ

元からある構文を全てを含む例として、前述のプログラムを型エラースライサで実行し、結果が得られるまでの過程を説明する。複数の集合が記述されている箇所は、部分的に抽象化した際に型エラーになる要素が現れたため、その構文内で再帰されたことを表す。

1. let 文：
   `[let sum = □ in ...; let sum = ... in □]`
2. fun 文：
   `[fun x -> □]`, `[fun y -> □]`
3. 関数適用：
   `[x □ y; □ + y; x + □]`,`[□ □ y; □ + □]`
4. 関数適用：
   `[□ 1 2.0; sum □ 2.0; sum 1 □]`,
   `[□ □ 2.0; sum □ □]`
5. 型エラースライスが決定：
   `let sum x y = □ + y in sum □ true`

7 評価

以下の基準に従って、本研究で追加した構文の例と 6 節の例に対する評価を行う。

- 質問回数の変化
- 質問が必要かつ最低限かどうか
- 残った式が必要かつ最低限かどうか

7.1 6 節の例

抽象化された `x` と `1` に対する質問全てが削減された。これら以外に対する質問のみで必要かつ最低限といえるが、`x` が抽象化されたことによって `sum` が `'a -> int -> int` 型となり、エラーメッセージの表示に不都合が生じる。

7.2 レコード

```
before: {a = 1; b = "2"; c = 3; }
 after: {a = □; b = "2"; c = □; }
```

型エラースライスによって、3 つの部分式のうち 2 つが抽象化されて、質問回数が 3 回から 1 回に減少した。質問は `b` の要素に対して行われる。スライスによって抽象化された `c` でも型の衝突は発生していたが、`b` を修正した後に再び型エラーデバッガを実行すれば、抽象化されずに質問が行われるので特に問題はない。よって、上の例は全ての基準を満たしていて、型エラースライスが十分に有用であるといえる。

7.3 コンストラクタ

(a) 要素数が定義と一致している場合

```
before: X (1, "2", 3)
 after: X (□, "2", □)
```

(b) 要素数が定義と一致していない場合

```
before: X (1, 2)
 after: X (□, □)
```

(a) はレコードの場合とほぼ同様である。(b) は型エラースライスによって、全ての部分式が抽象化されて、質問なしで原因を特定できるようになった。よって、いずれの例も全ての基準を満たしていて、型エラースライスが十分に有用であるといえる。

7.4 パターンマッチ

```
before: match m with p1 -> n1 | p2 -> n2
```

(a) m と p1,p2 で型が衝突する場合

```
after: match m with p1 -> □ | p2 -> □
```

(b) p1 と p2 で型が衝突する場合

```
after: match □ with p1 -> □ | p2 -> □
```

(c) n1 と n2 で型が衝突する場合

```
after:match □ with p1 -> n1 | p2 -> n2
```

(a) はパターンマッチの結果、(b) は全ての部分式、(c) はパターンマッチする式と、いずれも型の衝突とは関係のない部分式が抽象化されて、該当箇所の数だけ質問回数も減少した。しかし、型の衝突と関係のないパターンは抽象化されず、無駄な質問が残ったままである。

8 まとめと今後の課題

本研究では、対馬らの型エラースライサの構文を拡張し、実装を行なった。これにより、追加した全ての構文で型エラースライスによる効果が得られることを確認できた。一方、型がより一般的になったことによる不都合が生じたり、パターンに対する質問の削減が行われないといった問題点があった。

今後の課題は二点挙げられる。ひとつは、型がより一般的になったことによる不都合の解消である。もうひとつは、型エラースライサのパターンへの対応である。さらに、これらを行った後に型エラースライスの効果を機械的に測定したい。

参考文献

[1] O. Chitil. Compositional explanation of types and algorithmic debugging of type errors. *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP'01)*, pp. 193–204, September 2001.

[2] E. Y. Shapiro. *Algorithmic Program Debugging*. Cambridge: MIT Press, 1983.

[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の手法の提案. コンピュータソフトウェア, Vol. 30, No. 1, pp. 180–186, 2013.

[4] 対馬かなえ, 浅井健一. 重み付き型エラースライスの提案. コンピュータソフトウェア, Vol. 31, No. 4, pp. 131–148, 2014.