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[1,a)]　対馬 かなえ[2,b)]　浅井 健一[1,c)]

概要：OCaml のような静的な型付けを行う言語では，型によって多くの実行時エラーを取り除いている．一方，型エラーがあるとユーザが自身でデバッグを行い，正しい型付けになるようプログラムを書き換える必要がある．これはユーザの負担となるため，可能な限りプログラミング支援ツールが補助するべきである．本論文では，既存研究の型エラーの原因範囲を狭める型エラースライサを拡張・実装し，対話的にユーザへ質問しながらエラー箇所を特定する型エラーデバッガに追加した．型エラースライスの導入により質問回数を削減することができた．一方，質問範囲が削減されることにより，型がより一般的になるため，ユーザへのわかりやすさが削減される等の問題も存在することなどの問題点も判明した．

キーワード：型エラー，デバッグ，型エラースライス，OCaml，関数型言語

1. はじめに

ML や Haskell などの強い型付き関数型言語では，自動的な型付け（型推論）によって多くの実行時エラーを取り除いている．しかし型が正しく付くプログラムを書くことは容易ではない．また，コンパイラは型エラーに関してエラーメッセージを返すが，エラーメッセージが常に型エラーの原因を指摘するとは限らず，多くの場合，プログラマは自分で型エラーの原因を探すことになる．

1.1 型エラーとその原因

具体例として，OCaml で書かれた次の型エラーを含むプログラムと OCaml のコンパイラが返すエラーメッセージを考えてみる．

```
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しようとすると，プログラム中で下線部が引かれた部分で型の衝突が起こり，型エラーが発生する．エラーメッセージは，2.0 には int 型が期待されたが，2.0 自体は 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依って無数に考えられる．例えば，プログラマが考える型エラーの原因が 2.0 にある場合には，このエラーメッセージは適当であり，プログラマはエラーを理解して修正することが出来る．しかし，型エラーの原因が違う箇所にある場合には，なぜか 2.0 に int 型が期待されたという情報しか得られないため，プログラマ自身で型エラーの原因を

[1] お茶の水女子大学
[2] 国立情報学研究所
[a)] g1320544@is.ocha.ac.jp
[b)] k_tsushima@nii.ac.jp
[c)] asai@is.ocha.ac.jp

探す必要がある．

そもそも型エラーが発生する原因は，二つの単一化できない型がプログラム中に存在しているためである．前の例では，2.0 と + という二つの部分の型が衝突している．それに加えて，二つの型が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型エラーが起こる．

コンパイラは前述のプログラムについて，推論器を用いて図 1 に示すような型推論木を生成する．図中の (a) は let 文に対する型推論木，(b) は in 以下の式に対する型推論木，(c) は (a) と (b) を部分木として持つプログラム全体の型推論木を表している．let 文による関数定義は，指定された変数名に関数が与えられたものだと解釈される．

これらの推論木から，float 型の 2.0 を sum の第二引数として渡しているが，sum の定義内で第二引数 y は + に渡されているため，int 型となる必要があることが明らかである．そのため二つの型が衝突することになった．このように，型エラーは衝突する二つの型を持つ部分と，それらが同一でなければならないと定める箇所によって引き起こされる．

1.2 型エラーデバッガとその問題点

Chitil [1] は合成的な型推論を行うことにより，対話的な型エラーデバッガを可能とした．対馬ら [5] はそれをコンパイラの型推論 器を用いて行う手法を提案した．それにより，実装の負担を減らし，実用的な言語を対象とした実装を可能になった．型エラーデバッガでは，プログラマへの質問を行うことにより，対話的にエラー箇所を特定する．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，プログラマが 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．プログラマがそのように意図していれば y (yes)，そうでなければ n (no) と答えればよい．この質問の回答によって，次に質問される箇所が変化する．

質問の内容は最汎型木という特殊な木を元に作られている．最汎型木では，他の部分の型に影響されない箇所に最も一般的な型が与えられている．型エラーデバッガでは図 2 を使用している．通常の型推論木では，一番上の段で変数の型は x: int ， y: int になるが，最汎型木ではそれぞれの部分式が最も一般的な型を持つ点が異なる．これによって，原因が x や y ではなく，+ にあることが特定可能になる．

型エラーの原因となっている箇所の特定には，アルゴリズミックデバッギング [3] を利用している．アルゴリズミックデバッギングは，Shapiro によって考案された木構造を持つもののエラー箇所を特定するための手法で，以下の順序でエラーの箇所を特定する．

(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どうかを見る．
(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．

質問は原因として特定された箇所の子ノードの意図を尋ねることから始める．複数の子ノードがある場合，プログラム中でより手前に記述されている箇所を優先する．プログラマの意図通りであれば，型エラーデバッガの内部で該当箇所にエラーがないと判断し，共通の親を持つ他のノードに対して順にアルゴリズミックでバッギングを行い，対象となり得る他のノードがなければ，そのノードに原因があると特定される．一方，プログラムの意図通りでなければ，該当箇所にエラーがあると判断し，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．また，環境が存在する際は，先に環境の意図を尋ねる．

図 2 の木を用いて，デバッグすることを考え

$$\dfrac{\dfrac{\dfrac{\Gamma_{\{x:int,y:int\}} \vdash \mathrm{x} : int \quad \Gamma_{\{x:int,y:int\}} \vdash + : int \to int \to int \quad \Gamma_{\{x:int,y:int\}} \vdash \mathrm{y} : int}{\Gamma_{\{x:int,y:int\}} \vdash \mathrm{x} + \mathrm{y} : int}}{\Gamma_{\{x:int\}} \vdash \mathrm{fun\ y} \to \mathrm{x} + \mathrm{y} : int \to int}}{\Gamma \vdash \mathrm{fun\ x} \to \mathrm{fun\ y} \to \mathrm{x} + \mathrm{y} : int}}{\Gamma \vdash \mathrm{let\ sum\ x\ y} = \mathrm{x} + \mathrm{y} : int \to int \to int}$$

(a) let 文 に対する型推論木

$$\dfrac{\Gamma_{\{sum:int\to int\to int\}} \vdash \mathrm{sum} : int \to int \to int \quad \Gamma_{\{sum:int\to int\to int\}} \vdash 1 : int \quad \Gamma_{\{sum:int\to int\to int\}} \vdash \boxed{2.0 : \mathrm{float}}}{\Gamma_{\{sum:int\to int\to int\}} \vdash \mathrm{sum\ 1\ 2.0} : error}$$

(b) in 以下の式に対する型推論木

$$\dfrac{\text{(a) let 文に対する型推論木} \quad \text{(b) in 以下の式に対する型推論木}}{\Gamma \vdash \mathrm{let\ sum\ x\ y} = \mathrm{x} + \mathrm{y\ in\ sum\ 1\ 2.0} : error}$$

(c) 全体の型推論木

図 1: コンパイラの型推論木

$$\dfrac{\dfrac{\dfrac{\Gamma_{\{x:\alpha,y:\beta\}} \vdash \mathrm{x} : \alpha \quad \Gamma_{\{x:\alpha,y:\beta\}} \vdash + : int \to int \to int \quad \Gamma_{\{x:\alpha,y:\beta\}} \vdash \mathrm{y} : \beta}{\Gamma_{\{x:int,y:int\}} \vdash \mathrm{x} + \mathrm{y} : int}}{\Gamma_{\{x:int\}} \vdash \mathrm{fun\ y} \to \mathrm{x} + \mathrm{y} : int \to int}}{\Gamma \vdash \mathrm{fun\ x} \to \mathrm{fun\ y} \to \mathrm{x} + \mathrm{y} : int}}{\Gamma \vdash \mathrm{let\ sum\ x\ y} = \mathrm{x} + \mathrm{y} : int \to int \to int}$$

(a) let 文 に対する最汎型木

$$\dfrac{\Gamma_{\{sum:int\to int\to int\}} \vdash \mathrm{sum} : int \to int \to int \quad \Gamma_{\{sum:int\to int\to int\}} \vdash 1 : int \quad \Gamma_{\{sum:int\to int\to int\}} \vdash \boxed{2.0 : \mathrm{float}}}{\Gamma_{\{sum:int\to int\to int\}} \vdash \mathrm{sum\ 1\ 2.0} : error}$$

(b) in 以下の式に対する最汎型木

$$\dfrac{\text{(a) let 文に対する最汎型木} \quad \text{(b) in 以下の式に対する最汎型木}}{\Gamma \vdash \mathrm{let\ sum\ x\ y} = \mathrm{x} + \mathrm{y\ in\ sum\ 1\ 2.0} : error}$$

(c) 全体の最汎型木

図 2: 型エラーデバッガによる最汎型木

る．型エラーの原因は，まず型の衝突があった `sum 1 2.0` から開始する．この結果を元に，この関数適用のどこかに原因があると仮定し，子ノードについて質問を行う．はじめの質問に y と回答した場合には，型エラーデバッガは `sum` が型エラーの原因ではないと判断し，次に `sum` の引数について尋ねる．一方，n と回答した場合には，型エラーデバッガは `sum` が型エラーの原因であると判断し，上の定義部分に移動し，次に `sum` の定義について尋ねる．

対馬らの型エラーデバッガの問題点は，原因になり得ない箇所についても質問が行われることである．前述のプログラムの場合，型エラーの発生に `1` と `sum` の第一引数 `x` は関与していないので，これらに対する質問は無駄であるといえる．また，プログラムに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ちうる値が含まれると，すべての型のフィールドや引数について尋ねることになり，無駄な質問が多くなる．

1.3 型エラースライス

このような問題は型エラースライスを導入することで解決される．型エラースライスとは，型エラーに関係する部分のみを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ようになる．

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

□ という部分は，2.0 と + の型の衝突に関係しておらず，省かれた部分である．型エラースライスは一つの型の衝突に関係する全ての部分を含むため，その中に必ず型エラーの原因が含まれる．

1.4 提案手法についての説明

型エラーに関係ない質問を省略すればプログラマの負担が減り，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るか否かと同義である．そこで，本論文では対馬ら [4] の型エラースライサの実装および構文を拡張し，それによって抽象化された部分に対する質問を省略する．

前述のプログラムを実行した際の，型エラーデバッガの質問の推移を木構造で表現したものを図 3 に示す．図中の各ノードは，葉ノードが型エラーの原因であると特定された箇所，それ以外が下線部について質問されていることを意味する．また，下線部に含まれる変数について質問されている場合，対象の変数を枠で囲っている．ノードの深さは，それに到達するまでにした質問の回数となり，質問に対して y と回答したら左側の子ノードを，n と回答したら右側の子ノードをたどる．型エラースライスの導入前後によって，×に置き換えられたノードは，型エラースライサの導入によって省略されるべき箇所である．質問の省略は，プログラマの意図を尋ねる必要がなく，型エラーデバッガが自動的に y と解釈して良い箇所で行われる．そのため，省略されたノードの右部分木に含まれる質問も全て省略される．

2. 型エラースライサ

本節では，対馬らの型エラースライサをコンストラクタ，レコード，パターンマッチで拡張している．ベースとなる型エラースライサについては，対馬らの論文 [4] を参照せよ．拡張した構文を図 4 と図 5 に示す．

2.1 アルゴリズムの概要

新しく追加した構文を含む例について，型エラースライスを得ることを考える．

2.1.1 コンストラクタの型エラースライス

まず，コンストラクタ X で 2 番目の引数の型が int ではなく，別の型である下記の例について考える．

```
type x = X of int * string * int
X (1,2,3)
```

はじめにコンストラクタの構文全体に注目し，引数を部分的に抽象化した集合を考える．引数はタプルとして与えられているので，抽象化の結果はタプルに準ずる．タプルを抽象化した集合を全てコンストラクタとして置き換えて，これらを型推論器にかけると，

```
X (□, 2, 3), X (1, 2, □)
```

が型エラーになる．それをさらに部分的に抽象化した集合を型推論器にかけるという過程を繰り返すと，

```
X (□, 2, □)
```

の時点でこれ以上抽象化すると型エラーになる要素が現れなくなり，最終的に

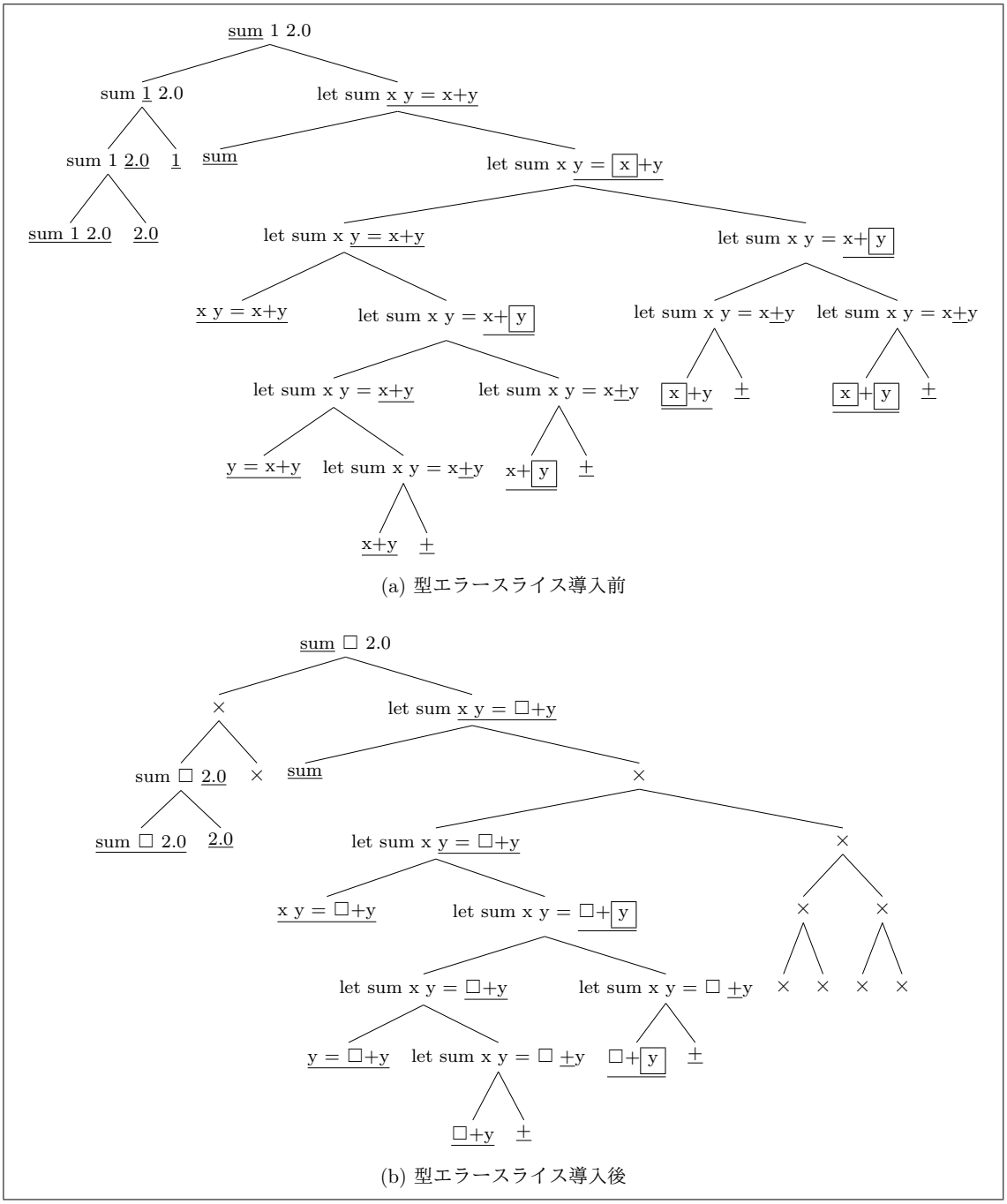
```
X (□, 2, □)
```

が得られる．

次に，コンストラクタ Y の定義上の引数の数が 1 個でない下記の例について考える．

```
type y = Y of bool * bool
Y (true)
```

はじめにコンストラクタの構文全体に注目し，引数を部分的に抽象化した集合を考える．その集合は引数を抽象化したコンストラクタを唯一の要素として持ち，型推論器にかけると型エラーになる．これ以上抽象化はできないが型エラーになるので，



(a) 型エラースライス導入前

(b) 型エラースライス導入後

図 3: 型エラーデバッガの質問の推移

$$
\begin{array}{lcll}
(l : label) & := & \text{(プログラムにおける当該式の} & \\
 & & \text{開始位置と終了位置の組.} & \\
 & & \text{プログラム中で一意.)} & \\
(f_n : field) & := & \text{(レコード型の各要素を表す名前.} & \\
 & & \text{プログラム中で一意.)} & \\
(M : term) & := & C^l & \text{(コンストラクタ)} \\
 & | & C^l M & \text{(コンストラクタ)} \\
 & | & \{f_1 = M; & \\
 & & \vdots & \\
 & & f_n = M\} & \text{(レコード)} \\
 & | & (match\ M\ with & \\
 & & | P \rightarrow M & \\
 & & \vdots & \\
 & & | P \rightarrow M) & \text{(パターンマッチ)} \\
(\tau : type) & := & C^l & \text{(コンストラクタ型)} \\
 & | & C^l of \tau & \text{(コンストラクタ型)} \\
 & | & \{f_1 : \tau; & \\
 & & \vdots & \\
 & & f_n : \tau\} & \text{(レコード型)}
\end{array}
$$

図 4: ラムダ計算の構文と型（拡張部分）

$$
\begin{array}{lcll}
(S : slice) & := & C^l & \text{(コンストラクタ)} \\
 & | & C^l S & \text{(コンストラクタ)} \\
 & | & \{f_1 = S; & \\
 & & \vdots & \\
 & & f_n = S\} & \text{(レコード)} \\
 & | & (match\ S\ with & \\
 & & | P \rightarrow S & \\
 & & \vdots & \\
 & & | P \rightarrow S) & \text{(パターンマッチ)} \\
(P : pattern) & := & c^l & \text{(定数)} \\
 & | & x^l & \text{(変数)} \\
 & | & (P, \ldots, P)^l & \text{(組)} \\
 & | & C^l & \text{(コンストラクタ)} \\
 & | & C^l P & \text{(コンストラクタ)} \\
 & | & \{f_1 = P; & \\
 & & \vdots & \\
 & & f_n = P\} & \text{(レコード)}
\end{array}
$$

図 5: スライスの構文（拡張部分）

最終的に

```
Y(□)
```

が得られる．

2.1.2 レコードの型エラースライス

レコード r のフィールド f3 の要素が，string 型ではなく char 型である下記の例について考える．

```
type r =  {f1 : string;
           f2 : string;
           f3 : char; }
{f1 = "abc"; f2 = "de"; f3 = "f"}
```

はじめにレコードの構文全体に注目し，引数を部分的に抽象化した集合を考える．その集合は

```
{f1 = □; f2 = "de"; f3 = "f"},
{f1 = "abc"; f2 = □; f3 = "f"},
{f1 = "abc"; f2 = "de"; f3 = □ }
```

となる．これらを型推論器にかけると，先頭の要素が型エラーになり，それをさらに部分的に抽象化した集合を型推論器かけるという過程を繰り返すと，

```
{f1 = □; f2 = □; f3 = "f"}
```

の時点でこれ以上抽象化すると型エラーになる要素が現れなくなり，最終的に

```
{f1 = □; f2 = □; f3 = "f"}
```

が得られる．

2.1.3 パターンマッチの型エラースライス

まず，下記のパターンマッチをする式とパターンの型が合わない例について考える．

```
match 1 with true -> 1 | false -> 0
```

はじめにパターンマッチの構文全体に注目し，各式を部分的に抽象化した集合を考える．その集合は

```
match □ with true -> 1 | false -> 0,
match 1 with true -> □ | false -> 0,
match 1 with true -> 1 | false -> □
```

となる．これらを型推論器にかけると，先頭の要素は型エラーにならず，その次の要素で型エラーになる．それをさらに部分的に抽象化した集合を型推論器かけるという過程を繰り返すと，

`match 1 with true -> □ | false -> □`
の時点でこれ以上抽象化すると型エラーになる要素が現れなくなり，最終的に
`match 1 with true -> □ | false -> □`
が得られる．

次に，下記のパターン同士で型が合わない場合について考える．

`match 1 with 1 -> 1 | false -> 0`

はじめにパターンマッチの構文全体に注目し，各式を部分的に抽象化した集合を考える．その集合は

`match □ with 1 -> 1 | false -> 0,`
`match 1 with 1 -> □ | false -> 0,`
`match 1 with 1 -> 1 | false -> □`

となる．これらを型推論器にかけると，先頭の要素が型エラーになり，それをさらに部分的に抽象化した集合を型推論器かけるという過程を繰り返すと，全ての式が抽象化される．これ以上抽象化はできないが型エラーになるので，最終的に
`match □ with true -> □ | false -> □`
が得られる．

さらに，下記のパターンの結果同士で型が合わない場合について考える．

`match 1 with 1 -> 1 | 0 -> false`

はじめにパターンマッチの構文全体に注目し，各式を部分的に抽象化した集合を考える．その集合は

`match □ with 1 -> 1 | 0 -> false,`
`match 1 with 1 -> □ | 0 -> false,`
`match 1 with 1 -> 1 | 0 -> □`

となる．これらを型推論器にかけると，先頭の要素が型エラーになる．これ以上抽象化すると型エラーになる要素が現れなくなり，最終的に
`match □ with 1 -> 1 | 0 -> false`
が得られる．

2.2 プログラム

図 6 に対馬らの型エラースライサを元に拡張したプログラムを示す．

abst_one は受け取ったスライスの一箇所を抽象化したスライスの集合を返す関数である．無限ループを避けるために，集合には抽象化した結果が元と同じであるようなものは含めない．コンストラクタを受け取った場合，引数が複数（タプル）であれば引数について再帰し，単数であれば引数を抽象化する．レコードを受け取った場合，タプルの場合と同様に各要素を抽象化し，末尾にスケルトンを加える．パターンマッチの場合，はじめにパターンマッチをする式を抽象化し，それからパターンマッチの各結果を抽象化する．

search は拡張前と同じである．この関数はより抽象化されたスライスの候補集合と，そのコンテキスト，サンク（実行を一時中断させたプログラムを指し，必要に応じて引数を渡すことで実行を再開する）を受け取る．候補の中にコンテキストのもとで型エラーになるものが存在する場合には，その式をより抽象化するために *get_slice* を呼び出す．存在しない場合にはサンクを呼び出す．このサンクによって構文を残しつつ部分式の抽象化を行っている．

get_slice は型エラースライスを求めるメイン関数である．□に適用すると型がつくコンテキスト *cxt* のもとで型エラーになるスライス S を受け取り，より抽象化された型エラースライスを一つ返す．コンストラクタを受け取った場合，*search* に渡すサンクの中で，コンストラクタの引数について再帰する．レコードを受け取った場合，関数適用やタプルと同様に各要素を置き換えるようなコンテキストの元でのスライスを順に求めて，最後にそれらをレコードとしてひとつにまとめて返す．パターンマッチの場合，パターンマッチをする式を置き換えるようなコンテキストの元でのスライスを求めた後，同様にパターンマッチの結果のスライスを順に求める．

2.3 実装

拡張した型エラースライサは，OCaml 4.02.1 の型エラーデバッガ上で実装した．その際，if，try，constraint 等の構文にも対応させている．複数行に渡るプログラムの場合には，型の衝突が発生するまでの環境下で，型の衝突が発生した行から順にさかのぼって各行の型エラースライスを求めている．また，字句解析，構文解析，型推論等は OCaml のものを再利用している．

$$
\begin{array}{lll}
abst\_one & : & slice \to slice\ list \\
abst\_one[\![C^l S]\!] & = & [C^l \square] \setminus C^l S \\
abst\_one[\![C^l(S_1, \ldots, S_n)^l]\!] & = & [C^l(\square, \ldots, S_n)^l; \ldots; C^l(S_1, \ldots, \square)^l; \\
 & & C^l(@\ S_1 \ldots S_n)] \setminus C^l(S_1, \ldots, S_n)^l \\
abst\_one[\![\{f_1 = S_1; \ldots, f_n = S_n\}]\!] & = & [\{f_1 = \square; \ldots, f_n = S_n\}; \ldots; \\
 & & \{f_1 = S_1, \ldots, f_n = \square\}] \setminus \{f_1 = S_1; \ldots; f_n = S_n\} \\
abst\_one[\![match\ S_0\ with & & \\
\quad |P \to S_1| \ldots |P \to S_n]\!] & = & [match\ \square\ with\ P \to S_1| \ldots |P \to S_n; \\
 & & \ match\ S_0\ with\ P \to \square| \ldots |P \to S_n; \\
 & & \ match\ S_0\ with\ P \to S_1| \ldots |P \to \square; \\
 & &] \setminus (match\ S_0\ with\ P \to S_1| \ldots |P \to S_n) \\
 & & \\
search & : & (slice\ list * (slice \to slice) * (unit \to slice)) \to slice \\
 & & \text{(省略：拡張前と同じ)} \\
 & & \\
get\_slice & : & (slice * (slice \to slice)) \to slice \\
get\_slice[\![(C^l S, cxt)]\!] & = & search(abst\_one[\![C^l S]\!], cxt, (fun\ () \to C^l(get\_slice[\![(S, cxt)]\!]))) \\
get\_slice[\![(\{f_1 = S_1; & & \\
\quad \ldots; f_n = S_n\}, cxt)]\!] & = & search(abst\_one[\![(S_1, S_2, \ldots, S_n)^l]\!], cxt, (fun\ () \to \\
 & & let\ S'_1 = get\_slice'[\![(S_1, (fun\ y \to cxt(y, S_2, \ldots, S_n)^l))]\!]\ in \\
 & & let\ S'_2 = get\_slice'[\![(S_2, (fun\ y \to cxt(S'_1, y, \ldots, S_n)^l))]\!]\ in\ \ldots \\
 & & let\ S'_n = get\_slice'[\![(S_n, (fun\ y \to cxt(S'_1, S_2, \ldots, y)^l))]\!]\ in \\
 & & (S'_1, S'_2, \ldots, S'_n)^l) \\
get\_slice[\![(match\ S_0\ with & & \\
\quad |P \to S_1| \ldots |P \to S_n, cxt)]\!] & = & search(abst\_one[\![match\ S_0\ with \\
 & & |P \to S_1| \ldots |P \to S_n]\!], cxt, (fun\ () \to \\
 & & let\ S'_0 = get\_slice'[\![(S_0, (fun\ y \to cxt(match\ y\ with \\
 & & |P \to S_1| \ldots |P \to S_n)))]\!]\ in \\
 & & let\ S'_1 = get\_slice'[\![(S_1, (fun\ y \to cxt(match\ S'_0\ with \\
 & & |P \to y| \ldots |P \to S_n)))]\!]\ in \ldots \\
 & & let\ S'_n = get\_slice'[\![(S_n, (fun\ y \to cxt(match\ S'_0\ with \\
 & & |P \to y| \ldots |P \to y)))]\!]\ in \\
 & & match\ S'_0\ with\ |P \to S'_1| \ldots |P \to S'_n)) \\
 & & \\
get\_slice' & = & \text{(省略：拡張前と同じ)}
\end{array}
$$

図 6: 型エラースライサ（拡張部分）

3. 評価

本節では型エラースライス導入による効果について，型エラーの原因として特定される可能性がある構文と，実際の授業でのプログラムに対して評価を行う．評価の基準は以下の三点である．

- 型エラースライス導入前後での質問回数の変化（構文の評価では，すべての質問に y と回答した場合）
- 型エラースライス導入後の質問内容が必要かつ最低限かどうか
- 型エラースライスで抽象化されずに残った情報が必要かつ最低限かどうか

3.1 関数適用

下記のプログラムとその型エラースライスについて考える．

```
プログラム：
(fun x y z -> x + y + z) 1 2.0 3.0
型エラースライス：
(fun x y z -> x + y + z) □ 2.0 □
```

このプログラムの質問回数は型エラースライス導入前が 4 回，導入後が 2 回である．プログラム中の下線は，質問で聞かれる箇所である．質問は関数部分（fun x y -> x + y + z）と 2 番目の引数 2.0 に対して行われる．型エラースライスによって型の衝突とは関係ない引数に対する質問が省略された．ただし，複数の引数で型の衝突が発生している場合には，最も手前に現れるものだけが残り，それ以降は全て省略される．

型エラーデバッガが示すエラーメッセージは，石井 [2] らによって初心者向けに改良されたもので，以下のように関数部分，全ての引数の型，型の衝突が発生している箇所で要求される型が含まれる．

```
Error17 : 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，引数，2 番目の引数に要求される型は以下の通りです．
関数部分 fun x  -> fun y  -> fun z  ->
(x + y) + z :
int -> int -> int -> int
 1 番目の引数の型：int
 2 番目の引数の型：float
 3 番目の引数の型：int
 2 番目の引数に要求される型：int
```

型エラースライサを使用すると，1 番目の引数の型が 'a，2 番目の引数の型 が 'b に置き換えられる．これらは型エラースライスを求める際に抽象化されて，□に置き換えられた箇所である．そのため，型の衝突と関係ない箇所でより一般的な型が表示されるが，エラーメッセージには，型の衝突に関係ある式の型があれば十分である．

第 1 節で例として示したプログラムもこの構文の例に該当する．型エラースライスによって質問回数が減り，型の衝突とは関係ない箇所の質問が省略されるが，□となった部分の型を表示できなくなる．

3.2 コンストラクタ

あらかじめ下記のような型が定義されていると仮定する．

```
type tree = Empty
          | Node of tree * int * tree
```

対馬らの型エラーデバッガで型エラーの原因がコンストラクタにあると特定されるようなプログラムは，コンパイラが示すエラーによって大まかに二通りに分類された．ひとつはコンストラクタの引数の数が定義と異なることが原因によるエラーで，この場合型の衝突がコンストラクタ型の式全体で発生する．もうひとつはコンストラクタの引数の数は定義と同じだが，引数の型が定義と異なることが原因によるエラーで，この場合型の衝突が型コンストラクタ型の式の中で発生する．

3.2.1 式全体での型の衝突

下記のプログラムとその型エラースライスについて考える．

```
プログラム：
Node (Empty, 1, 2, Empty)
型エラースライス：
Node (□, □, □, □)
```

このプログラムの質問回数は型エラースライス導入前が 4 回，導入後が 0 回である．プログラム中の下線は，質問で聞かれる箇所である．型エラースライスによって Node のすべての引数が省かれたため，質問を一切せずに型エラーの原因が特定された．型エラーの原因であると特定された式の引数の数は得られるが，各引数の型は得られなくなる．エラーメッセージには，特定された箇所での各引数の型と，定義上の各引数の型を含むのが最良である．

3.2.2 式の中での型の衝突

下記のプログラムとその型エラースライスにつ

いて考える．

```
プログラム：
Node (Empty, true, [])
型エラースライス：
Node (□, true, □)
```

このプログラムの質問回数は型エラースライス導入前が 3 回，導入後が 1 回である．プログラム中の下線は，質問で聞かれる箇所である．質問は 2 番目の引数 true に対して行われる．型エラースライスによって型の衝突とは関係ない引数に対する質問が省略された．ただし，複数の引数で型の衝突が発生している場合には，最も手前に現れるものだけが残り，それ以降は全て省略される．この例では [] が tree の定義と衝突しているが，型エラースライスでは抽象化されている．型エラーの原因であると特定された箇所以外の型は，tree 型が型エラーが発生する以前に定義されているため，エラーメッセージに定義上の引数の型を含められるが，型の衝突があったにも関わらず抽象化された箇所の型は得られなくなる．この現象は関数適用で見られたものと同様で，エラーメッセージには，型の衝突に関係ある式の型と定義上の各引数の型を含めれば十分である．

3.3 レコード

あらかじめ下記のような型が定義されていると仮定する．

```
type diary = {
  title : string;
  date : int * int * int;
  text : string; }
```

対馬らの型エラーデバッガで型エラーの原因がレコードにあると特定されるようなプログラムのうち，フィールドの数とそれぞれの名前はすべて正しいが，要素の型が定義と異なることが原因によるエラー（型の衝突が式の中で発生する場合）のみを対象とする．

下記のプログラムとその型エラースライスについて考える．

```
プログラム：
{title = "abc";
 date = 2017;
 text = None; }
型エラースライス：
{title = □;
 date = 2017;
 text = □; }
```

このプログラムの質問回数は型エラースライス導入前が 3 回，導入後が 1 回である．プログラム中の下線は，質問で聞かれる箇所である．質問はフィールド date の要素 2017 に対して行われる．型エラースライスによって型の衝突とは関係ない要素に対する質問が省略された．ただし，複数の要素で型の衝突が発生している場合には，最も手前に現れるものだけが残り，それ以降は全て省略される．この例では None が diary の定義と衝突しているが，型エラースライスはで抽象化されている．型エラーの原因であると特定された箇所の引数以外の型は，diary 型が型エラーが発生する以前に定義されているため，定義上の引数の型は得られるが，型の衝突があったにも関わらず省略された箇所の型は得られなくなる．この現象は関数適用で見られたものと同様で，エラーメッセージには，型の衝突に関係ある式の型と定義上の各要素の型を含めれば十分である．

3.4 パターンマッチ

対馬らの型エラーデバッガでは，型が衝突する箇所によってパターンマッチが原因のエラーを複数に分けている．ここでは，型の衝突が発生している箇所によって

- パターンマッチをする式とパターン
- パターン同士
- パターンマッチの結果同士

の三通りを対象とし，これらをパターンの式によって

- 全て定数
- 定数以外を含む

の二通りに分けて評価を行う．

3.4.1 パターンマッチをする式とパターンでの型の衝突

まず，全てのパターンが定数である下記のプログラムとその型エラースライスについて考える．

```
プログラム：
let x = 1.0 in
  match x with
  | 1 -> 1.0 | 2 -> 2.0 | 3 -> 3.0
  | _ -> raise Not_found
型エラースライス：
let x = 1.0 in
  match x with
  | 1 -> □ | 2 -> □ | 3 -> □
  | _ -> □
```

このプログラムの質問回数は型エラースライス導入前が 4 回，導入後が 1 回である．プログラム中の下線は，質問で聞かれる箇所である．質問はパターンマッチをする式 x に対して行われる．型エラースライスによって型の衝突とは関係ない全てのパターンの結果に対する質問が省略された．パターンの結果は型の衝突とは無関係なので，そもそもエラーメッセージに含まれていない．

次に，パターンに定数以外を含む下記のプログラムとその型エラースライスについて考える．

```
プログラム：
let lst = 0 in
match lst with
  [] -> false
| first :: rest -> true
型エラースライス：
let lst = 0 in
match lst with
  [] -> □
| first :: rest -> □
```

このプログラムの質問回数は型エラースライス導入前が 4 回，導入後が 2 回である．プログラム中の下線は，質問で聞かれる箇所である．質問はパターンマッチをする式 lst とパターンにある変数 rest について行われる．このプログラムではパターンの型が 'a list となり，リストの要素の型は何でも良いので first に対する質問は行われない．型エラースライスによって全てのパターンマッチの結果が省略されたが，パターンに対する質問は省略されずに残っている．型エラーの原因の特定には，first :: rest というパターンが 'a list 型であるという情報があれば十分なので，このプログラムではパターンに対する質問が行われないのが最良である．

以上より，式に対しては型エラースライスによる効果が見られたが，パターンに対しては無駄が残っているといえる．

3.4.2 パターン同士での型の衝突

まず，全てのパターンが定数である下記のプログラムとその型エラースライスについて考える．

```
プログラム：
let x = 1 in
  match x with
  | 1 -> 1.0 | 2 -> 2.0 | 3.0 -> 3.0
  | _ -> raise Not_found
型エラースライス：
let x = □ in
  match □ with
  | 1 -> □ | 2 -> □ | 3.0 -> □
  | _ -> □
```

このプログラムの質問回数は型エラースライス導入前が 3 回，導入後が 0 回である．プログラム中の下線は，質問で聞かれる箇所である．型エラースライスによってプログラム中の式全てが省かれたため，質問を一切せずに型エラーの原因が特定される．型エラースライスの導入前後でメッセージの変化は見られず，パターンマッチする式とパターンの結果は型の衝突とは無関係なので，そもそもエラーメッセージに含まれていない．

次に，パターンに定数以外を含む下記のプログラムとその型エラースライスについて考える．

プログラム：
```
let find d = match d with
  None -> false
|{title = a; date = b; text = c} -> true
```
型エラースライス：
```
let find d = match □ with
  None -> □
|{title = a; date = b; text = c} -> □
```

このプログラムの質問回数は型エラースライス導入前が 5 回，導入後が 3 回である．プログラム中の下線は，質問で聞かれる箇所である．質問はパターンにある各変数 a，b，c について行われる．型エラースライスによってプログラム中の式全てが省略されたが，定数でないパターンに対する質問は省略されずに残っている．型エラーの原因の特定には，そのパターンが diary 型であるという情報があれば十分なので，このプログラムではパターンに対する質問が行われないのが適切である．

以上より，式に対しては型エラースライスによる効果が見られたが，パターンに対しては無駄が残っているといえる．

3.4.3 パターンマッチの結果同士での型の衝突

まず，下記のすべてのパターンが定数であるプログラムとその型エラースライスについて考える．

プログラム：
```
let x = 1 in
  match x with
  | 1 -> 1.0 | 2 -> 2.0 | 3 -> 3.0
  | _ -> Not_found
```
型エラースライス：
```
let x = □ in
  match □ with
  | 1 -> 1.0 | 2 -> □ | 3 -> □
  | _ -> Not_found
```

このプログラムの質問回数は型エラースライス導入前が 4 回，導入後が 2 回である．プログラム中の下線は，質問で聞かれる箇所である．質問ははじめのパターンの結果 1.0 と型の衝突が発生したパターンマッチの結果 Not_found に対して行われる．型エラースライスによって型の衝突とは関係ないパターンマッチする式とパターンマッチの結果に対する質問が省略される．型エラースライスの導入前後でメッセージの変化は見られず，エラーメッセージには省略されずに残った 2 つの型が含まれる．

次に，下記のパターンに定数以外を含むプログラムとその型エラースライスについて考える．

プログラム：
```
let tmp t = match t with
| Node (Empty, n1, r1) -> 0
| Node (l2, n2, Empty) -> 1
| Node (l3, n3, r3) -> 2
| Empty -> Not_found
```
型エラースライス：
```
let tmp t = match □ with
| Node (Empty, n1, r1) -> 0
| Node (l2, n2, Empty) -> □
| Node (l3, n3, r3) -> □
| Empty -> Not_found
```

このプログラムの質問回数は型エラースライス導入前が 11 回，導入後が 9 回である．プログラム中の下線は，質問で聞かれる箇所である．質問はパターンに含まれる各変数と，パターンマッチの結果のうち 0 と Not_found に対して行われる．型エラースライスによって型の衝突とは関係ないパターンマッチする式とパターンマッチの結果が省略されたが，定数でないパターンに対する質問は省略されずに残っている．全てのパターンが型の衝突とは関係ないので，このプログラムではパターンに対する質問が行われないのが最良である．

以上より，式に対しては型エラースライスによる効果が見られたが，パターンに対しては無駄が残っているといえる．

3.5 授業でのプログラムの例

下記のプログラムとその型エラースライスについて考える．これは関数型言語の授業で実際に学生が書いたプログラムである．型エラーが発生

した時点でプログラムの実行を停止するため，型エラースライスに test2 と test3 が含まれていない．

```
プログラム：
(* 目的：二次方程式の係数を与えられたら，判別式の値を返す *)
(* hannbetsusiki :
     int -> int -> int -> int *)
let hannbetsusiki a b c = b*b-4*a*c

(* テスト *)
let test1 = hannbetsusiki 1 2 3 = (-8)
let test2 = hannbetsusiki 2 4 5 = (-24)
let test3 = hannbetsusiki 1 5 2 = 17

(* 目的：二次方程式の係数を与えられたら，解の個数を返す *)
(* kai_no_kosuu :
    int -> int -> int -> int *)
let kai_no_kosuu a b c =
  if hannbetsusiki a b c > 0 then "2"
  else if hannbetsusiki a b c < 0
  then "0" else "1"

(* テスト *)
let test1 = kai_no_kosuu 1 2 3 = 0
let test2 = kai_no_kosuu 1 4 4 = 1
let test3 = kai_no_kosuu 1 5 2 = 2

型エラースライス：
let hannbetsusiki a b c = □
let test1 = □
let test2 = □
let test3 = □
let kai_no_kosuu a b c =
  if □ then □
  else if □
  then □ else "1"
let test1 = kai_no_kosuu □ □ □ = 0
```

この時，OCaml のコンパイラは以下のようなエラーメッセージを示し，2 には string 型が期待されたが，2 自体は int 型であると指摘している．型の衝突は，= は左右で引数の型が同一でならなければならないと強制することにより発生する．

```
let test1 = kai_no_kosuu 1 2 3 = 0
Error: This expression has type int but
an expression was expected of type
string
```

このプログラムでは，型エラースライス導入前後で以下のような変化が見られた．

- kai_no_kosuu が string 型を返す関数だと意図していない場合，各引数に対する質問が省略される．
- kai_no_kosuu の定義で，各引数の型に対する質問が省略される．
- if 文の条件部分と then 部分の式に対する質問が省略される．

関数適用 kai_no_kosuu 1 2 3 は型の衝突に関係あるが，その式自体に型エラーはない．そのため，関数が最終的に返す型は残されるが，引数は抽象化の対象となり，kai_no_kosuu がより一般的な型となる．kai_no_kosuu の定義では，if 文の条件部分は型の衝突とは関係がなく，条件部分全体が抽象化されるので，hannbetsusiki も全体が抽象化される．また，test1 の定義によって then 部分の式を抽象化しても型エラーになるので，then 部分の式も抽象化の対象となる．= では質問が省略されなかったが，二つの引数の型は多相だが同一でなければならないという制約によって抽象化できなかったが，結果として正しい．

以上より，型エラースライス導入による効果は確認できたが，kai_no_kosuu の各引数の型を表示できなくなる点は問題である．

4. まとめ

本論文では対馬ら [4] の型エラースライサの構文をコンストラクタ，レコード，パターンマッチで拡張子，実装を行った．これによる利点は質問回数の削減である．型の衝突が発生する可能性がある全ての構文で，型エラースライス導入による効果

が得られた．一方，質問範囲が削減されて型がより一般的になったことで，エラーメッセージの表示に不都合が生じる場合があった．この現象は主に複数の引数を取る構文で見られる．また，型エラースライスの対象となる構文は，現時点では式のみで，パターンは含まれていない．そのため，パターンマッチではパターンに対する質問が削減されず，無駄な質問が多く残っている場合があった．

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．一般的に熟練者は多相を理解しているため，より一般的な型を見せても問題ないが，初心者のためには対応が必要となる．これには一部の型情報を残したまま最小ではない型エラースライスを作ることが考えられ，抽象化されずに残すべき場所についての考察が必要となる．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．パターンは型規則を持つと同時に束縛でもあるため，式のように単純に□に置き換えることができない．これにはパターンを分解して，型制約を外して束縛だけを残す処理が必要になる．

参考文献

[1] Chitil, O.: 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 (2001).

[2] Ishii, Y. and Asai, K.: Report on a User Test and Extension of a Type Debugger for Novice Programmers, *3rd International Workshop on Trends in Functional Programming in Education, Electronic Proceedings in Theoretical Computer Science*, Vol. 170, pp. 1–18 (2014).

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

[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. 29, No. 1, pp. 78–95 (2012).

[5] 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).