# コンパイラの型推論を利用した型デバッグ手法の提案

対馬 かなえ [1], 浅井 健一 [2]

お茶の水女子大学

[1] tsushima.kanae@is.ocha.ac.jp [2] asai@is.ocha.ac.jp

**概 要** 本論文ではコンパイラに備わっている型推論をそのまま使い、型デバッガを作成する手法について述べる。これまでの型デバッガでは、型推論を型デバッガ内で独自に行っていた。その問題点として、独自の型推論とコンパイラの型推論との間で推論結果に不一致が起こる可能性がある点が挙げられる。また型推論の複雑さは実装の難しさと直結するため、対象とする言語が大きいほど実装するのは難しくなる。

そこで本研究では、デバッグ対象のプログラムを分割し、それぞれを既存の型推論にかけることによって、型デバッグを行う手法を提案する。ある一定の性質を満たすように分割したプログラムそれぞれを型推論にかけ、その結果が正しいかどうかを、アルゴリズミックデバッギングを使用してプログラマへ対話的に問いかけることで型エラーの原因となった箇所を見つける。これにより、型推論を自分で実装することなく型デバッガを実装することが出来る。詳細には、value restriction を含む多相な型に対応し、Caml Light で実装を行った。

## 1 はじめに

強い型付き言語では、型検査によって多くのプログラム実行時エラーを取り除くことが出来ている。特に ML や Haskell などの強い型付き関数型言語では、型の多相性によって型に自由度を与えつつ、型の注釈がないプログラムに対して自動的な型付け (型推論) を行っている。これはプログラマによる注釈なしに汎用的な型を求め、かつ安全性を確保しているという点で非常にすばらしい。しかし、型が正しいプログラムを書くことは容易ではなく、プログラマはコンパイル時に型エラーを指摘されて、修正すべき箇所を探すためにデバッグを行うことが多い。また、コンパイラが指摘する箇所がプログラマが実際に修正したい箇所と離れていることも頻繁に起こる。

型エラーの修正に関して考えてみると、OCaml や Haskell のような広く使用されている言語全てをサポートする型デバッガは少ない。そのため、プログラマはコンパイラのエラーメッセージを頼りに、自身で推論しながら型デバッグを行うことになる。しかし型のデバッガがあれば、プログラマの型エラーを探すという負担を軽減出来る。そのような型デバッガは何をプログラマに示すべきだろうか。プログラムの修正を行う際にプログラマが知りたいのは、自分が考えているプログラムへ直すためにプログラム中で直すべき箇所と、どのように直すべきかである。コンパイラのエラーメッセージでそれらを指摘することが出来るだろうか。

ここで、次のような型エラーが生じるプログラムについて考えてみよう。

```
let rec f g lst = match lst with
  | [] -> []
  | fst :: rest -> (g fst) :: (f g rest) in
(f 1 [1;2;3]) @ [5;6;7]
```

このプログラムで、型の不一致が起こる原因となった部分は、プログラム中の枠で囲まれた二カ所である。関数 f の第一引数 g は、(g fst) では関数として、(f 1 [1;2;3]) では int として使われている。関数型 a -> b と整数型 int を同じ型にすることは出来ないため、型推論で型エ

ラーが指摘される。このプログラムを OCaml コンパイラ version 3.12.1 にかけると、以下のようなエラーメッセージが得られる。

(f 1 [1;2;3]) @ [5;6;7] (* 下線部分がエラーメッセージの "This expression" にあたる *)

```
Error: This expression has type int but an expression was expected of
type 'a list -> 'b
```

このエラーメッセージは、(f 1 [1;2;3]) の第一引数 1 の型は int であるが、'a list -> 'b であるべきだと指摘している。しかし、これが常にプログラマにとって親切なエラーメッセージであるとは限らない。もしプログラマが関数 f は数字 g と数字のリスト lst を受け取り、lst の全ての要素に g を足す関数だと考えてプログラムを書いたとすると、修正すべき箇所は (f 1 [1;2;3]) という関数適用ではなく、(g fst) である。つまり、プログラマの意図した f の内容によって修正すべき箇所は以下のように変わる。

| | プログラマの意図した f の内容 | 修正すべき箇所 |
|---|---|---|
| (1) | map (関数 g をリストの全ての要素へ適用する関数) | (f 1 [1;2;3]) 中の 1 |
| (2) | add_g (数字 g をリストの全ての要素へ足す関数) | (g fst) |

そもそもコンパイラのエラーメッセージは、型推論中に型の不一致が起きた場所を指摘しているだけであり、型エラーの原因となった修正すべき箇所を指摘するものではない。そして、上に示したように全く同じプログラムの全く同じ型エラーに関しても、プログラマがどのようなプログラムを意図していたかによって、修正すべき箇所は異なる。従って一つのエラーメッセージだけで、常にプログラマが修正すべき箇所を示すことは残念ながら不可能である。

ここで、型エラーはなぜ起こるのか考えてみる。例えば上の例では、以下のようにプログラマの意図した f の型が異なる。

| | プログラマの意図した f の型 | エラー修正の一例 |
|---|---|---|
| (1) | ('a -> 'b) -> 'a list -> 'b list | 1 ⇒ (fun x -> x + 1) |
| (2) | int -> int list -> int list | (g fst) ⇒ (g + fst) |

(1) のケースでは、本来関数を渡さなくてはならない箇所に数字を渡していたことが型エラーの原因である。よってエラー箇所の 1 は推論された型は int だが、プログラマが意図したその部分の型は関数である。(2) のケースでは、"+" という演算子を書き忘れたことが原因である。それによって (g fst) の g が関数として扱われ、f の型がプログラマの意図と異なってしまったことが分かる。これらが示すように特定したいエラー箇所では、プログラマが意図した型と型推論が推論した型が異なる。よって、プログラマの意図した型を受け取り、型推論が推論した型と異なる箇所を探すことによって、型エラーの特定が出来る。

本論文の流れは以下の通りである。まずエラー箇所を特定するためにはどのような木が必要かを考察する (2 節)。そして、エラー箇所を特定出来る型推論木を作る既存研究とその問題点を示し、それらの問題点を解決する手法を提案する (3 節)。提案手法ではコンパイラの型推論を使用することで、独自の型推論を作ることなくエラー箇所が特定可能な型推論木を作る。それによる提案手法の利点は以下の通りである。

- コンパイラの型推論を使用しているため、コンパイラの型推論に則している
- 型推論を実装する必要がないため、OCaml や Haskell 等の大きい言語に対する型デバッガの実装が容易になる

まず提案手法を型付きラムダ計算に適用し (4 節)、その後多相型へ拡張する (5 節)。これらによって型エラーのあるプログラムに対してエラー箇所の特定が可能となるが、型機構によって

はより細やかな指摘が望まれることがある。その例として OCaml などで採用されている value restriction に対応した拡張を行う (6 節)。また対象言語を Caml Light として実装を行ったため、実装の詳細とその際に使用した改良、実行例を示す (7 節)。最後に型エラーに関連する研究との比較を行い (8 節)、本論文をまとめて今後の課題について述べる (9 節)。

# 2 問題点

## 2.1 通常の型推論で得られる型推論木

プログラマの意図と比較してエラー箇所を特定するためには、型推論木が必要となる。しかし既存のコンパイラの型推論を適用して得られる型推論木ではデバッグを行うことが出来ない。本節ではその理由について述べる。ここで `(fun x -> x + x) true` という型エラーのあるプログラムを例として考えてみる。このプログラムを通常の型推論にかけた時に得られるであろう型推論木は以下の通りである。

```
{x:int} ⊢x:int  {x:int} ⊢+:int -> int -> int  {x:int} ⊢x:int
--------------------------------------------------------------
                     {x:int} ⊢x + x:int
             ----------------------------------
             {} ⊢(fun x -> x + x):int -> int         {} ⊢ true:bool
             ------------------------------------------------------
                         (fun x -> x + x) true··· 型エラー
```

ここでプログラマが考える正しいプログラムは `(fun x -> x and x) true` とする。するとプログラマは関数部分の型は `bool -> bool` と考えているが、型推論木中では `int -> int` 型となっているので、上の型推論木の枠線部分が間違っていることが分かる。しかし、具体的にその中のどの部分が間違っているのかは特定出来ない。例えばこの型推論木では、最上段の `x` の型が型の単一化によって `int` となっている。しかし `x` というプログラム単体を考えると、`int` となる必要はない。これらの型推論結果は、このプログラム全体を通しての型になっていて、お互いに影響を及ぼしている。このように一部分の型が独立にその部分だけから推論される形になっていない点がエラー箇所が特定出来ない理由である。

## 2.2 合成的な型推論

つまりプログラムの部分の型がそこだけで決まる形になっていれば、その型推論木を使用してデバッグを行うことが出来る。これを実現するために Chitil は合成的な型推論 [1] を考案した。具体的には、プログラムの部分部分の judgement (型環境と型) を求め、それらを合成していくことで全体の judgement を求める。上に示したプログラムに対して合成的な型推論を適用すると、次のような型推論木を得ることが出来る。

```
③ {x:a} ⊢ x:a    {} ⊢ +:int -> int -> int  ③{x:a} ⊢ x:a
---------------------------------------------------------
               ② {x:int} ⊢ x + x:int
           ------------------------------------
         ① {} ⊢ (fun x -> x + x):int -> int            {} ⊢true:bool
           ---------------------------------------------------------
                        (fun x -> x + x) true··· 型エラー
```

この型推論木では、最上段の `x` の型が `int` となっていない。よって一部分の型が独立にその部分だけから推論される形になっているため、エラー箇所が特定出来る。

合成的な型推論がどのように型推論を行うかという概要を述べる。まず最上段の 3 つのプログラムの型をそれぞれ独立に推論する。そのとき、“x” というプログラム単体はどんな型でも良いため、{x:a} という環境の下で a 型であると型付けされる。次にそれらの型推論結果を合成する形で “x + x” の型推論を行う。その際に最上段の x の型はそのまま保たれている。合成的な型推論の特徴は、下段でどのような単一化や環境の変化が起きたとしても、上段までその影響が及

ばないことである。これによってプログラムの部分の型が独立にそこだけで決まる形になる。このような木を求めると、前節の型推論木とは違い型エラーの原因を特定することが可能になる。

この型推論木でデバッグを行うことを考えてみる。プログラマの考えている正しいプログラムが先ほどと同じく `(fun x -> x and x) true` だとすると、同様の理由から ① より上の部分にエラーがあることが分かる。次に ② では `x` の型が `int` であるため、`x` は `bool` であるというプログラマの意図と異なる。そして ③ の {`x:a`} ⊢ `x:a` はその意図と矛盾しないため、正しい。よって型エラーの原因は枠で囲まれた部分 `x + x` にあることが分かる。

我々は先行研究 [10] でこの手法を OCaml のサブセットで実装し、その改良を行った。この型推論木を使用した型デバッガを実現すると、上で示したプログラムでは以下のような質問がなされ、それらに返答するとエラー箇所が特定される。

```
(fun x -> x + x) の型は int -> int ですか？    (質問<1>)
> いいえ
(x + x) において x の型は int ですか？          (質問<2>)
> いいえ
+ の型は int -> int -> int ですか？            (質問<3>)
> はい
型エラーの原因の箇所が特定されました：(x + x)
```

プログラマが意図した型を答えて行くだけで、エラー箇所が正しく特定出来ていることが分かる。このような型デバッガではプログラマ自身が推論してエラー箇所を見つける必要がなく、意図した型を答えるだけでエラーの箇所が特定されるため、非常に便利であり使いやすい。

### 2.3 問題点

合成的な型推論ではコンパイラの型推論とは別に独自の型推論を行う必要がある。独自の型推論は、より細やかな型情報の収集などが可能になるという利点があるが、欠点も存在する。

一点はコンパイラの型推論の推論結果と不一致が起こる可能性があることである。コンパイラの型推論には様々な工夫が施されているため、推論結果を完全に一致させるのは困難である。もしコンパイラのバージョンが変更され、型推論が変わったならばそれに合わせて修正する必要がある。また、コンパイラの型推論と同じ結果になると示すことも難しい。もう一点は、独自の型推論の複雑さは型デバッガの実装の難しさと直結するため、対象とする言語が大きいほど実装するのは難しくなることである。前述の我々の実装 [10] では OCaml の基本的な構文をカバーしたが、実装までに多くの時間を費やした。特に OCaml の Objective な部分までサポートする実装を考えると非常に困難であることが予想される。

Chitil は合成的な型推論によって部分独立な型推論木を実現したが、必要なのは部分の型がそこだけで決まるかどうかという性質だけであり、合成的に型推論を行う必要はない。そこで本研究では、型エラーを起こすデバッグ対象のプログラムを適切に分割し、それぞれを既存のコンパイラの型推論にかけることによって、部分独立な型推論木を作成し、型デバッグを行う手法を提案する。これによりコンパイラの型推論との不一致は生じなくなり、大きな言語に対しても実装が簡単になると考えられる。

## 3 提案手法の概要

本節では既存のコンパイラの型推論を用いて、前節で示した部分独立な型推論木を作る手法の概要を提案する。そしてエラー箇所を求めるために用いる手法、アルゴリズミックデバッギングを説明する。これは合成的な型推論 [1] でもエラー箇所を特定するために使用された。

### 3.1 概要

プログラムの部分の型を求めるには、プログラムを部分プログラムに分割し、それぞれを型推論にかければよい。例えば $M$ の部分プログラムが $M_1$ と $M_2$ である場合、$M_1$ と $M_2$ を独立にコンパイラの型推論にかければ、下に示すようにそれぞれの部分だけに依存した型を求められる。

$$\frac{M_1 \quad M_2}{M} \quad \Rightarrow \quad \frac{M_1 : \tau_1 \quad M_2 : \tau_2}{M : \tau}$$

どのように分割するかは 4 節以降で述べるが、本節ではこのようなプログラムの分割が出来るとした場合、どのようにエラー箇所を特定するかについて述べる。

### 3.2 アルゴリズミックデバッギング

型推論木を使ってエラー箇所を特定するためにアルゴリズミックデバッギングを用いる。アルゴリズミックデバッギングは Prolog のプログラムのエラー箇所を特定するために Shapiro によって提案された [7]。その後、実行時エラーの特定 [6] やプログラムの意味的エラーの特定 [8] など幅広く使われているが、本質的に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ある。概要としては、木のそれぞれの部分に関して正否 (oracle) を受け取り、その正否に従って木の中を移動しながらエラーの箇所を特定する。正否に関してはユーザの入力を使うことが多い。

具体的なアルゴリズムとしては、あるノードがエラーであった場合には、その子ノードがエラーであるかどうかを見る。そして、全ての子ノードが正しければ、その親ノードがエラーの原因と特定される。もし子ノードにエラーが見つかった場合には、そのまた子ノードに対してアルゴリズミックデバッギングを行う。

2.2 節の型デバッガによる質問は下に示した推論木に対してアルゴリズミックデバッギングを行った結果であった。ここで、アルゴリズミックデバッギングがどのようにエラー箇所を特定しているのかを見て行く。

$$\dfrac{\text{①}\dfrac{\text{②}\dfrac{\text{③}\{\texttt{x:a}\} \vdash \texttt{x:a} \quad \text{④}\{\} \vdash \texttt{+:int -> int -> int} \quad \text{③}\{\texttt{x:a}\} \vdash \texttt{x:a}}{\{\texttt{x:int}\} \vdash \texttt{x + x:int}}}{\{\} \vdash \texttt{(fun x -> x + x):int -> int}} \qquad \{\} \vdash \texttt{true:bool}}{\texttt{(fun x -> x + x) true} \cdots \text{型エラー}}$$

まず一番下のノードは型エラーなので、上の木のどこかにエラーがあるはずである。

① 2.2 節の 質問<1>より `(fun x -> x + x)` の型は `int -> int` ではない ⇒ 木を登る
② 質問<2> より `x + x` の `x` の型は `int` ではない ⇒ 木を登る
③ どちらも自明に成り立つので質問を省略
④ 質問<3> より正しい ⇒ 全ての子ノードが正しいため、エラー箇所は ② に特定される

このように推論木に対してアルゴリズミックデバッギングを適用し、プログラマからの回答を使って推論木の中を移動して行くことで、エラー箇所が特定出来る。

### 3.3 プログラムの分割が満たすべき性質

アルゴリズミックデバッギングを使用して正しくエラー箇所を見つけるためには、プログラムを分割する際に以下の性質を満たさなくてはならない。

$$\mathcal{T}(ExFun_\sigma[\![M]\!]) \Rightarrow \forall(M', \sigma') \in Div[\![M]\!]\sigma,\ \mathcal{T}(ExFun_{\sigma'}[\![M']\!])$$

$\mathcal{T}$ は正しく型がつくことを示す述語であり、$Div$ はプログラムの分割を意味する。また次節で詳しく述べるが $ExFun$ は $M$ の自由変数を全て展開するような変換である。ここでプログラムとは式 $M$ とその自由変数の集合 $\sigma$ の組 $(M, \sigma)$ とする。(そのように定義する理由は次節で後

$$
\begin{array}{rcll}
(M : term) & ::= & c & \text{(定数)} \\
 & | & x & \text{(変数)} \\
 & | & \lambda x.M & \text{(関数抽象)} \\
 & | & M_1\ M_2 & \text{(関数適用)} \\
(\tau : typ) & ::= & b & \text{(基底型)} \\
 & | & \tau_1 \to \tau_2 & \text{(関数型)}
\end{array}
$$

**図 1.** ラムダ計算の構文と型

$$
\begin{array}{rcl}
fvs & = & var\ list \\
Div : term \to fvs & \to & (term * fvs)\ list \\
Div[\![c]\!]\sigma & \to & [] \\
Div[\![x]\!]\sigma & \to & [] \\
Div[\![\lambda x.M]\!]\sigma & \to & [(M, x :: \sigma)] \\
Div[\![M_1\ M_2]\!]\sigma & \to & [(M_1, \sigma); (M_2, \sigma)]
\end{array}
$$

**図 2.** 分割 $Div$

$$
\begin{array}{rcl}
ExFun : fvs \to term & \to & term \\
ExFun_{[]}[\![M]\!] & = & M \\
ExFun_{a::r}[\![M]\!] & = & \lambda a.ExFun_r[\![M]\!]
\end{array}
$$

**図 3.** 自由変数の展開 $ExFun$

$$
\begin{array}{rcl}
C : fvs \to typ & \to & ((var * typ)\ list * typ) \\
C_{[]}[\![\tau]\!] & = & ([], \tau) \\
C_{a::r}[\![\tau_1 \to \tau_2]\!] & = & \text{let } (\mu, \tau) = C_r[\![\tau_2]\!] \text{ in} \\
 & & (\mu[a \to \tau_1], \tau)
\end{array}
$$

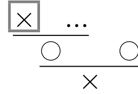
**図 4.** 環境の回収 $C$

```
let t = typing (ExFun_fv[[exp]]) in
let (env, t') = C_fv[[t]] in
  (env, t')
```

**図 5.** *judgement*(exp, fv)

述する。) 上の性質が意味するのは、正しく型が付くプログラム $(M,\sigma)$ を分割した全てのプログラム $(M',\sigma')$ は正しく型がつく必要があるということである。逆にある部分プログラム $(M', \sigma')$ に型がつかない場合、その部分プログラムを含むプログラム $(M, \sigma)$ にも型がつかない。

この性質が満たされない場合、あるプログラムには型がつくがその一部は型がつかないということが起きるため、次のような型推論木が存在出来る。



この木の真の型エラーの原因は枠で囲まれた最上段の × である。しかしこの木にアルゴリズミックデバッギングを適用すると、原因として特定されるのは最下段の × である。なぜなら中段の ○ を見た時点で、その内部は正しいものとされるため、最上段の × まで質問されることがない。これでは正しく型エラーの原因が特定されたとは言えない。よって正しくエラー箇所を特定するためには、分割の際に上で述べた性質を満たす必要がある。

## 4 型付きラムダ計算

本節では型付きラムダ計算を対象として、コンパイラの型推論を使って部分独立な型推論木を作るために必要な、分割などのアルゴリズムを定義する。図 1 に対象言語とするラムダ計算の構文 $M$ と型 $\tau$ を示した。式 $M$ は、定数 $c$ か、変数 $x$、関数抽象 $\lambda x.M$、関数適用 $M_1\ M_2$ である。また型 $\tau$ は `int` などを含む基底型 $b$ か、関数型 $\tau_1 \to \tau_2$ である。

本論文でのコンパイラの型推論とは、OCaml や Caml Light で採用されているような、プログラムを受け取って型を返すものを想定している。加えて、定義されていない変数 (自由変数) が現れた場合にはエラーを返すように実装されているとする。

### 4.1 プログラムの分割と自由変数の展開

プログラムの分割とは、大まかに言ってプログラムを部分プログラムに分解することである。$\lambda x.M$ ならば $M$ に、$M_1\ M_2$ ならば $M_1$ と $M_2$ となる。これらの部分プログラムを型推論にかけると、部分独立な型推論木が得られそうだが、$\lambda x.M$ の場合に問題が起こる。なぜなら $M$ の中には変数 $x$ が含まれている可能性があり、$\lambda x$ に束縛されない $M$ の中でその $x$ は自由変数となる。このような部分プログラム $M$ をコンパイラの型推論にかけると、変数 $x$ が定義されていないためエラーになってしまう。

そもそも $x$ はプログラムの一部を見ているために自由変数になっているが、実際にプログラム全体を見た時には単相で束縛されているはずで、それ自体は型に制限を持たない。よって単相になるような自由変数の型を得るために次のような手法をとる。まず、対象となる分割されたプログラム $M$ に、その自由変数の集合 $\vec{x}$ を関数抽象として展開し、$\lambda\vec{x}.M$ という形にして型推論を行う。型推論された型 $\vec{\alpha} \to \beta$ から、 $\vec{x}$ に対応する $\vec{\alpha}$ を回収することで、$M$ の型は $\vec{x}$ の型を $\vec{\alpha}$ とした下で $\beta$ であることが分かる。

以上の処理を行うために、プログラムにおける自由変数を求める必要がある。通常あるプログラムにおける自由変数は判定が出来るが、プログラムの一部だけを見た場合には判定が出来ない。例えば $fst$ という関数が言語上で初期から用意されている場合、$\lambda fst.M$ というプログラムの部分プログラム $M$ の中で、$M$ を見るだけではそれが初期から用意されている関数なのか、自由変数なのかが区別出来ない。そのため、プログラムとは式 $M$ とその自由変数の集合 $\sigma$ の組 $(M, \sigma)$ となる。プログラムの分割 $Div$ は図 2、自由変数の展開 $ExFun$ は図 3、環境の回収 $C$ は図 4 に示した。

本節での分割 $Div$ は前節に示した守るべき性質を満たしている。これは以下が成り立っていることから分かる。

$$\mathcal{T}(ExFun_{\sigma}[\![\lambda x.M]\!]) \Rightarrow \mathcal{T}(ExFun_{x::\sigma}[\![M]\!])$$

$$\mathcal{T}(ExFun_{\sigma}[\![M_1\ M_2]\!]) \Rightarrow \mathcal{T}(ExFun_{\sigma}[\![M_1]\!]) \wedge \mathcal{T}(ExFun_{\sigma}[\![M_2]\!])$$

従ってこの分割を使ってアルゴリズミックデバッギングを行えば、エラー箇所を正しく特定できる。

### 4.2 judgement

次にどのように judgement (型環境と型) を求めるのかを見て行く。まず、あるプログラム $M$ を $Div$ で分割し、部分プログラムと自由変数を求める。例えば部分プログラムが `f x` で、自由変数が `f` と `x` であったとすると、$ExFun$ を適用することで `fun f -> fun x -> f x` というプログラムを得ることが出来る。これによって通常は型がつかない自由変数を含むプログラムに対しても型推論を行えるようになる。次にそのプログラムをコンパイラの型推論にかけ、`('a -> 'b) -> 'a -> 'b` という型を受け取る。そして $C$ でその型から自由変数に対応するものを取り出すと、{`f:('a -> 'b), x:'a`} ⊢`f x:'b` が得られる。

あるプログラム `exp` の judgement を求める処理は図 5 に示した。`exp` を自由変数 `fv` で拡張し、型推論を行った後、自由変数の型環境を集めて、judgement を求めている。この例から得られた judgement を型推論木として示すと以下のようになる。

$$\frac{\{\texttt{f:('a -> 'b), x:'a}\} \vdash \texttt{f x:'b} \quad \ldots}{\ldots}$$

このような推論木をデバッグする際には、自由変数 `f` と `x` に関してもプログラマに質問する必要がある。なぜならば `f` が関数でないと思っているプログラマにとっても (式の内部構造を見なければ、ある式に `'b` という型がつくと言っているだけなので) `f x` は `'b` であることは正しい。これでは `f` が関数だと推論された原因は `f x` にあるが、`f x` は型エラーではないと判断されてし

まい、型エラーの原因は他の箇所にあるとされてしまう。よって、もしコンパイラの型推論において自由変数が型エラーとならずに適切に処理される場合にも、自由変数の型を上のような形で取得する必要がある。

### 4.3 デバッグ

ここまでで一つの式に対する judgement (型環境と型) を求めることが出来た。実際にデバッグを行うためには、型エラーのあるプログラム $(M, \sigma)$ を分割し、部分プログラムの集合 $\{(M', \sigma'), ..\}$ を得る。そして一つの部分プログラム $(M', \sigma')$ に対して judgement を求め、それがプログラマの意図に合っているかどうかを質問する。意図したものであった場合には、$(M', \sigma')$ 以外の分割されたプログラムに関して調べる。(そしてそれら全てが正しい場合には $(M, \sigma)$ が型エラーの原因と特定される。) 逆に意図したものでなかった場合には、$(M', \sigma')$ の中に型エラーが存在するので、$(M', \sigma')$ に対してアルゴリズミックデバッギングを再帰的に行う。このようにアルゴリズミックデバッギングを利用して型エラーの原因を特定することが出来る。

## 5 多相型の導入

本節では前節で定義したアルゴリズムを多相型へ拡張する。対象となる言語の構文を図 6 に示した。これは前節の構文に組、不動点演算子 $fix$、変数定義 $let$ を追加したものである。ML の構文にある関数定義 `let v v`$_1$`...v`$_n$ `= M`$_1$ `in M`$_2$ は $let\ v = \lambda v_1 ... v_n . M_1\ in\ M_2$ に、再帰関数 `let rec f v`$_1$`...v`$_n$ `= M`$_1$ `in M`$_2$ は $let\ f = fix\ f\ v_1 ... v_n \rightarrow M_1\ in\ M_2$ にそれぞれ対応する。

### 5.1 方針

`let a = 1 in let id = (fun x -> x) in (id 3, id true)` というプログラムを具体例として考えてみよう。4 節と同様に部分プログラムへ分割をすると、以下のようになる。

$$\frac{\{\}\vdash \texttt{1:int} \quad \{\}\vdash \texttt{(fun x -> x):'a -> 'a} \quad \texttt{(fun id ->(id a, id true))} \cdots \text{型エラー}}{\{\}\vdash \texttt{let a = 1 in let id = (fun x -> x) in (id a, id true):(int * bool)}}$$

このような分割では下段のプログラムは正しく型がつくが、上段の分割されたプログラムは型エラーとなる。これでは 3.3 節で示した性質を満たさないため、正しくエラー箇所を見つけることが出来ない。上段のプログラムが型エラーになった原因は、分割前のプログラムでは `id` は多相型として扱われているにも関わらず、分割したプログラムでは単相として扱われていることである。よって分割後にも多相型として扱える形で分割する必要があり、次のように分割すればよい。

$$\frac{\{\}\vdash \texttt{let a = 1 in let id = fun x -> x in id a:int} \quad \{\}\vdash \texttt{let a = 1 in let id = fun x -> x in id true:bool}}{\{\}\vdash \texttt{let a = 1 in let id = fun x -> x in (id a, id true):(int * bool)}}$$

この分割では `let` での変数定義はそのままに、`let` 文の本体部分のみを分割している。このような多相型へ拡張した分割は図 7 のように定義される。前節の分割との大きな違いは、let 環境 $\rho$ の導入である。これは後で `let` 文として展開するべきものを保存する環境である。多相を導入するとプログラムは式 $M$、自由変数 $\sigma$、let 環境 $\rho$ の組 $(M, \sigma, \rho)$ となる。このようにして多相に対応した分割を行うことで、正しく型がつく部分プログラムが得ることが出来る。

### 5.2 プログラムの分割と let 環境の保存

具体例として上の例 `let a = 1 in let id = (fun x -> x) in (id a, id true)` の分割を考える。$Div[\![let...]\!]$ が 2 回実行された時点で、let 環境 lenv は `[a` $\mapsto$ `(1, [], []); id` $\mapsto$ `((fun x` $\rightarrow$ `x), [], a` $\mapsto$ `(1, [], []))]` となる。そして本体部分の `(id a, id true)` が、`id a` と `id true` に分割される。よって分割後のプログラムは `(id a, [],` lenv`)` と `(id true,`

$$
\begin{array}{rcll}
(M : term) & = & c \mid x \mid \lambda x.M \mid M_1\ M_2 \mid (M_1, ..., M_n) & \text{(組)} \\
& \mid & fix\ f\ v_1\ ...\ v_n \to M & \text{(不動点演算子)} \\
& \mid & let\ v\ = M_1\ in\ M_2 & \text{(変数定義)} \\
(\tau : typ) & = & b \mid \tau_1 \to \tau_2 \mid \tau_1 * ... * \tau_n & \text{(直積型)}
\end{array}
$$

**図 6.** 多相な型を導入した構文と型

$$fvs = var\ \ list \qquad env = (var * (term * fvs * env))\ list$$

$$
\begin{array}{rcll}
Div : term \to fvs \to env & \to & (term * fvs * env)\ list & \\
Div[\![c]\!]\ \sigma\ \rho & \to & [] & \\
Div[\![x]\!]\ \sigma\ \rho & \to & [] & \text{if } x \in \sigma \\
Div[\![x]\!]\ \sigma\ \rho & \to & [\rho(x)] & \text{if } x \notin \sigma \wedge x \in \mathrm{dom}(\rho) \\
Div[\![\lambda x.M]\!]\ \sigma\ \rho & \to & [(M, x :: \sigma, \rho)] & \\
Div[\![M_1\ M_2]\!]\ \sigma\ \rho & \to & [(M_1, \sigma, \rho); (M_2, \sigma, \rho)] & \\
Div[\![(M_1,\ ...,\ M_n)]\!]\ \sigma\ \rho & \to & [(M_1, \sigma, \rho); ...; (M_n, \sigma, \rho)] & \\
Div[\![fix\ f\ v_1\ ...\ v_n \to M]\!]\ \sigma\ \rho & \to & [(M, [f; v_1; ...; v_n]@\sigma, \rho)] & \\
Div[\![let\ v\ = M_1\ in\ M_2]\!]\ \sigma\ \rho & \to & Div[\![M_2]\!]\ (\sigma \setminus [v])\ (\rho[v \mapsto (M_1, \sigma, \rho)]) &
\end{array}
$$

**図 7.** 分割 $Div$

let t = typing ($ExLet_{\mathtt{env}}(ExFun_{\mathtt{fv}}[\![\mathtt{exp}]\!])$) in
let (env, t’) = $C_{\mathtt{fv}}[\![t]\!]$ in
　(env, t’)

**図 8.** $judgement$(exp, fv, env)

$$
\begin{array}{rcl}
ExLet_{[]}[\![M]\!] & = & M \\
ExLet_{(v,(M_1,\sigma,\rho))::r}[\![M]\!] & = & ExLet_r[\![let\ v = M_1\ in\ M]\!]
\end{array}
$$

**図 9.** $let$ 環境の展開 $ExLet$

[], lenv) となる。judgement は図 8 のように求められる。まず 4 節と同様に $ExFun$ を使用して自由変数を展開し、その後、図 9 の $ExLet$ を使用して let 環境を `let` 文として展開する。`let` 文の展開は変数・関数定義の依存関係を守るために、環境に保存した順番で行われる。上に示した例に対する型推論木は次のようになる。

```
{x:'a}⊢let a = 1 in x:'a
----------------------------------
{}⊢let a = 1 in (fun x -> x):'a->'a
------------------------------------------------
{}⊢let a = 1 in let id = fun x -> x in id:'a->'a

                    {}⊢1:int
                    --------------------------------------------
                    {}⊢let a = 1 in let id = fun x -> x in a:int
---------------------------------------------------------------------
{}⊢let a = 1 in let id = fun x -> x in id a:int                   (略)
---------------------------------------------------------------------
{}⊢let a = 1 in let id = fun x -> x in (id a, id true):(int * bool)
```

`let` 文で定義された変数が $Div$ によって分割される場合 (図 7 の 3 つ目の定義にあたる) の let 環境には、その変数が定義された時の let 環境が使用される。それによって変数や関数定義の依存関係が保たれている。これは上の型推論木の枠線部分にあたり、関数 `id` の定義時に定義されていた `a` のみが展開されていることが分かる。自由変数に関しても、`let` の定義時の自由変数集合を用いることで、正しい自由変数集合を求めている。

let 環境に新しく追加する際には、自由変数集合から同名の変数を取り除くことによって正しい

束縛を保っている。自由変数集合に新しく追加される場合にも let 環境から同名の変数を取り除く必要があるように思われるが、その変数はプログラムの他の部分で使用されている可能性があるため取り除くことは出来ない。そこで judgement を求める際に、自由変数の展開を行ってから let 環境の展開を行うことで、正しく束縛されるようにしている。また、対象とするのは閉じたプログラムなので、変数の場合には let 環境か自由変数集合のどちらかに含まれる。

図 6 の構文には含まれていないが、相互再帰を含む構文に関しても上記とほぼ同じように簡単に拡張出来る。ただし相互再帰で同時に定義している関数では、同時に定義しているどれかの関数に型エラーの原因がある可能性が指摘された場合にも、全てに型エラーの可能性があるため、それを考慮して分割を行う必要がある。

本節の分割が満たすべき性質は以下に示すものである。

$$\mathcal{T}(ExLet_\rho(ExFun_\sigma[\![M]\!])) \Rightarrow \forall(M',\sigma',\rho') \in Div[\![M]\!]\ \sigma\ \rho,\ \mathcal{T}(ExLet_{\rho'}(ExFun_{\sigma'}[\![M']\!]))$$

これらの性質は式の構造に関する帰納法を使って示すことが出来るため、エラー箇所を正しく特定できる。また停止性に関しては、右辺の拡張後の構文木はいずれも、左辺の拡張後の構文木の部分木になっていることから保証される。

# 6 value restriction への対応

本節では OCaml や Caml Light などで、参照型を導入しつつ型安全を維持するために使われている value restriction への対応について考察する。図 10 に参照を導入した構文を示した。これまでの拡張でエラー箇所の特定は可能であるが、value restriction のような、途中で型を変更する型機構に関してはより細やかな指摘が望まれることがある。まず例として以下のような参照を使用したプログラムを考える。

```
let a = fun x -> x in let b = ref a in (!b 3, !b true)
```

value restriction とは上のようなプログラムにおいて、`b` を一度のみの具体化を許した弱い多相として扱うことである。上のプログラムにおいて `!b` の型は `'_a -> '_a` であるが (“`_`” は弱い多相型を示す)、`!b 3` を実行した時点で `int -> int` へと変化する。その後、`!b true` では `!b` の型がすでに `int -> int` になっているので型エラーとなる。なぜそのような制約を設けているかというと、参照型が入ってきた場合に任意の多相性を許すと型の健全性を崩してしまうためである。それを避けるために、`let` 文のうち式が値でないものに対しては弱い多相になるという制限を設けている。これが value restriction である。

上の例を前節のアルゴリズムに適用すると、以下の型推論木が得られる。

```
let a = ..  in let b = ..  in !b 3:int  let a = ..  in let b = ..  in !b true:bool
──────────────────────────────────────────────────────────────────────────────────
            let a = ..  in let b = ..  in (!b 3, !b true)··· 型エラー
```

この型推論木では、変数 `b` の型の変更が judgement に現れることがない。よって上のノード両方がプログラマによって正しいとされ、エラーは下のノードであると特定される。これは確かにエラーを含む部分を指摘している。しかし型の衝突の原因となった `b` の型の変更がプログラマに提示されることがないため、プログラマは原因となった箇所まで辿り着くことが出来ない。より細やかな指摘を行うためには型の変更が型推論木に現れるようにする必要がある。

そもそも value restriction の型は全体を通して単一的に扱われる。よって 4 節と同じように単相として扱う方法が考えられる。すると以下のような型推論木を作ることが出来る。

```
 {b:(int ->'_a) ref} ⊢ let a = ..  in !b 3:(int ->'_a) ref ->'_a
{b:(bool ->'_a) ref} ⊢ let a = ..  in !b true:(bool ->'_a) ref ->'_a
─────────────────────────────────────────────────────────────────────
         (fun b -> let a = ..  in (!b 3, !b true))··· 型エラー
```

$$
\begin{array}{rcll}
(M : term) & = & c \mid x \mid \lambda x.M \mid M_1\ M_2 \mid (M_1,\ ..., M_n) \mid ref\ M & \text{(参照定義)} \\
& \mid & !M & \text{(参照からの取り出し)} \\
& \mid & v := M & \text{(参照の書き換え)} \\
& \mid & fix\ f\ v_1\ ...\ v_n \rightarrow M \mid let\ v\ = M_1\ in\ M_2 & \\
(\tau : typ) & = & b \mid \tau_1 \rightarrow \tau_2 \mid \tau\ ref & \text{(参照型)}
\end{array}
$$

**図 10.** value restriction を含む構文と型

$$
\begin{array}{rcl}
& env = (var\ *\ (term\ *\ fvs\ *\ env))\ list & \\
Div : term \rightarrow env & \rightarrow & (term * fvs * env)\ list \\
Div[\![ref\ M]\!]\ \sigma\ \rho & \rightarrow & [(M,\sigma,\rho)] \\
Div[\![!M]\!]\ \sigma\ \rho & \rightarrow & [(M,\sigma,\rho)] \\
Div[\![v := M]\!]\ \sigma\ \rho & \rightarrow & [(M,\sigma,\rho)]
\end{array}
$$

**図 11.** 分割 *Div*

$$
\begin{array}{rcl}
ExVal : term & \rightarrow & term\ *\ var\ list \\
ExVal_{(v,(e,_,_))}[\![M]\!] & \rightarrow & \text{let (exp, lst)} = ExVal[\![M_2]\!]\ \text{in} \\
& & \text{if (is_nonexpansive } e\text{) then (exp, lst)} \\
& & \text{else } ((v, \text{exp}), v :: \text{lst}) \\
ExVal[\![M]\!] & \rightarrow & (M, [])
\end{array}
$$

**図 12.** value restriction が起きる変数の *ExVal*

$$
\begin{array}{rcl}
CVal : var\ list \rightarrow typ & \rightarrow & ((var * typ)\ list * typ) \\
CVal_{a::r}[\![(\tau_1 * \tau_2)]\!] & \rightarrow & \text{let (env, t)} = CVal_r[\![\tau_2]\!]\ \text{in } ((\text{a}, \tau_1) :: \text{env, t}) \\
CVal_{[]}[\![\tau]\!] & \rightarrow & ([], \tau)
\end{array}
$$

**図 13.** value restriction *CVal*

```
let (exp', vs) = ExVal_env[[exp]] in
let t = typing (ExLet_env[[ExFun_fv[[exp']]]]) in
let (env, t') = C_fv[[t]] in
let (env', t) = CVal_vs[[t']] in
    (env' @ env, t)
```

**図 14.** *judgement*(exp, fv, env)

このように扱うと、b の型が環境に現れるため、これらの b の型がプログラマの意図に反していた場合には、また型推論木を登ることで型エラーの原因を見つけることが出来る。しかし、このようにすると b の定義が let 文で提供されているにもかかわらず、それが let 環境に現れて来なくなってしまうので、b の中身がエラーの原因だった場合、そこに到達することができない。

そこで b のように value restriction によって型が具体化される可能性のあるものは let 環境と judgement の型環境の両方に入れることを考える。この方針に沿って型推論を行って得た型推論木を以下に示した。この型推論木では、b は let 環境に入っていると同時に型環境にその型が明示されている。このようにすると型環境で b の型の変化を把握しつつ、b の定義部分が間違っていた場合にもその中に入ってデバッグを行うことができるようになる。

$$
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\{\texttt{x:'a}\}\vdash\texttt{x:'a}}}{\{\}\vdash\texttt{fun x -> x:('a -> 'a)}}}{\{\}\vdash\texttt{let a = ..  in a:('a -> 'a)}}}{\{\}\vdash\texttt{let a = ..  in ref a:('_a -> '_a) ref}}}{\{\texttt{b:('_a -> '_a) ref}\}\vdash\texttt{let a = ..  in let b = ..  in b:('_a -> '_a) ref}}}{\{\texttt{b:('_a -> '_a) ref}\}\vdash\texttt{let a = ..  in let b = ..  in !b:('_a -> '_a)}} \quad (\text{略})}{\{\texttt{b:(int -> int) ref}\}\vdash\texttt{let a = ..  in let b = ..  in !b 3:int}} \quad (\text{略})}{\texttt{let a = ..  in let b = ..  in (!b 3, !b true)}\cdots\text{型エラー}}
$$

図 11 に value restriction で拡張した分割を示した。ほぼ前節の分割と同じであるため、加わった部分のみを示している。value restriction のための拡張で重要なのは図 14 の $judgement$ である。まず図 12 の $ExVal$ を使用して value restriction の対象となる変数の集合 $\overline{vs}$ を求め、対象となるプログラムとそれらが組になる式を求める。そして前節までと同様に、コンパイラの型推論を利用して型を求め、それから自由変数の型環境を求めている。最後に、図 13 の $CVal$ を使用して value restriction の対象となる変数の型環境を求める。それら二つの型環境を合わせることで、value restriction によって型が具体化される可能性があるものが let 環境 と judgement の型環境の両方に入ることになる。図 12 の $ExVal$ で使用されている関数 is_nonexpansive は value restriction の対象となるかどうかを返すコンパイラの関数であり、コンパイラによって value restriction の対象が異なる場合にも、同様の関数を使用することで対応出来ると考えている。

# 7 実装

ここまでを Caml Light を対象言語として実装した。デバッガ部分の行数は 400 行程度であり、コンパイラの型推論を利用しているが、型推論のプログラムには全く変更を加えていない。

## 7.1 改良

本論文では既存の型推論を使って部分独立な型推論木を作る手法を導入した。そのため部分独立な型推論木に対するデバッグの改良 [10] はそのまま適用でき、今回の実装にもプログラマの回答の再利用による改良を使用した。概略としては、アルゴリズミックデバッギングの際に木を登って行く場合、それまでに得たプログラマの回答を再利用する。あるプログラムで回答を得た場合、部分プログラムでは同じ型に推論されることが多いため、以前の回答を利用することで質問回数を減らすことが出来る。次節で示す実行例では、いくつかの質問がこの改良によって省略された。

## 7.2 実行例

実装では judgement から次の方針に則って質問を行った。

(1) まず、環境の中の変数の型が正しいかを尋ねる

(2) 全て正しかったら右辺の式の型が正しいかを尋ねる

どれか一つでも間違っていた時点で、その judgement は間違っていると判定する。

まず 3 節で示した単相型のみが出てくるプログラム `(fun x -> x + x) true` の実行例を示す。このプログラムをプログラマが考えている正しいプログラムが `(fun x -> x and x) true` としてデバッグを行った例を以下に示した。

```
The type of (fun x -> x + x) is int -> int?
> no
The type of x is int in ((+ x) x)?
```

```
> no
The type of + is int -> int -> int?
> yes
Error located! : (+ x)
```

これはプログラマの意図では x は int ではないのに int -> int -> int 型の + に関数適用したところがプログラマの意図に反していると指摘している。よってプログラマは関数部分か引数部分のどちらかを直す必要があり、この場合には関数部分を + から and へ修正する必要があると分かる。

次に多相型を含む以下のプログラムを考える。(これは合成的な型推論 [1] で例として使用されたものに類似した例である。)

```
let rec reverse lst = match lst with
| [] -> []
| fst :: rest -> (reverse rest) @ fst in (reverse [1;2;3]) @ [4;5;6]
```

このプログラムでは受け取ったリストの要素を逆順にしたリストを返すような (通常の) reverse 関数を定義している。よってプログラマは reverse 関数の型は 'a list -> 'a list だと考えている。しかし、(reverse rest) @ fst で fst をリストとして扱っている。(@ は二つのリストを結合する 'a list -> 'a list -> 'a list 型の関数である。) それによって reverse の型が 'a list list -> 'a list と推論され、型エラーが起きている。

```
The type of reverse is 'a list list -> 'a list?
> no
The type of lst in match expression is 'a list list?
> no
The type of [] is 'a list?
> yes
The type of reverse in ((reverse rest) @ fst) is 'a -> 'b list?
> yes
The type of fst is 'a list?
> no
The type of (@ (reverse rest)) is 'a list -> 'a list?
> yes
Error located! : (@ (reverse rest)) fst
```

上のデバッグでは (@ (reverse rest)) と fst の関数適用をしたところがプログラマの意図に反していると指摘されている。実際、上でプログラマが答えているように (@ (reverse rest)) の型は 'a list -> 'a list であり、fst の型は 'a list ではない。よってこの箇所が型エラーであり、本来は fst ではなく 'a list 型を持つ [fst] と書くべきであったことが分かる。

最後は value restriction を含む以下のような 6 節で示したプログラムである。

```
let a = fun x -> x in let b = ref a in (!b 3, !b true)
```

例えばプログラマが b は (int -> int) ref だと考えているとしてデバッグを行う。(実際には (!b 3, !b 1) のようなプログラムであったと考えられる。)

```
The type of b is (int -> int) ref in (!b 3)?
> yes
The type of (!b 3) is int?
> yes
```

```
The type of b is (bool -> bool) ref in (!b true)?
> no
The type of b is ('_a -> '_a) ref in !b?
> yes
The type of !b is ('_a -> '_a)?
> yes
The type of true is bool?
> yes
Error located! : !b true
```

このデバッグでは `!b true` が型エラーの原因として特定された。これはプログラマが `b` は `(int -> int) ref` と考えているが、`!b true` という関数適用の時点で弱い多相の型が変更され、意図しない型になったことを示している。

# 8 関連研究

エラーメッセージに関して、Wand [11] はエラーメッセージを改良するためのアルゴリズムを提案した。また型推論アルゴリズムの変更でもエラーメッセージの改良は行われており、Algorithm M [3] は、一般的に使われる Algorithm W よりも早く型の衝突を発見し、かつ広い箇所ではなく一部を型エラーとして指摘できるという点で優れている。しかし、1 節で示したように、エラーメッセージは役に立つことは多いが、一つのエラーメッセージで型エラーの箇所を指摘することは出来ない。

次に型デバッガを狭義に定義し、型エラーの原因や直すべき型を示すものとすると、2 節で挙げた部分独立な型推論木を作成する合成的な型推論 [1] がある。本研究はこの方針と同じようなデバッグを目指しているが、独自に型推論を行わずにコンパイラの型推論を利用することで、それらから生じる問題点を解決している。型デバッガを広義に定義し、型エラーの原因を見つけるために使うものとすると、型に関するスライスを使用する方法 [2] がある。これらの研究では型エラーが生じる原因となった部分のプログラムを切り出すことが出来、型エラーの原因を探す際にプログラマが全く情報を与えることなく、見るべき箇所が絞られるのが利点である。

型エラーの自動修正も提案されている。プログラムの構文を一部変更することにより、求められる正しく型がつくプログラムを修正案として提案するもの [4] がある。これは提案されたプログラムにプログラマの意図したプログラムがあった場合には、非常に有用である。

また、型エラーをデバッグする上で型を分かりやすく可視化することは重要である。これに関しては、型推論で生じた型エラーを可視化するためにグラフを使うもの [5] や型推論された型を表示するもの [9] などの研究が行われている。本研究にこれらの手法を取り込むことで、型デバッガからの質問がより分かりやすく示せるのではないかと考えている。

# 9 まとめと今後の課題

本論文ではコンパイラの型推論を利用して、部分独立な型推論木を作る手法を提案した。コンパイラの型推論を使うことによる利点は主に二つある。一つはこれまでの手法のように独自のデバッガの型推論を必要とせず、コンパイラの型推論を使用するため、型推論結果の不一致が起こらない点である。また独自の型推論の実装が必要がないため、OCaml や Haskell のような広く使用されている言語全てをサポートすることも可能になると考えられる。

具体的な手法としてはプログラムを分割し、既存の型推論を使ってそれぞれの部分プログラムの型を求め、部分独立な型推論木を作成した。詳細としては value restriction を含む多相な型に

対応し、OCaml の祖先 Caml Light を対象言語としてコンパイラの型推論を変更することなく実装を行った。

問題として挙げられる点は、対象言語の構文に応じて拡張したデバッガを実装する際には、それらがどう型付けされているかを良く知った上でプログラムの分割を作る必要があることが挙げられる。しかしそれは型機構に対応した型デバッガを実装しようとすれば当然避けられないことである。また糖衣構文等の型機構を持たないものに関しては、ほぼ単純に分解するだけでよく、今回の Caml Light を対象とした実装においてもそのように実装されている。

今後の拡張的な方向性としては、本研究で行わなかった型機構へ対応したい。具体的には OCaml で使用されている多相バリアントやオブジェクト、Haskell などで使用されているオーバーロードなどである。現時点では多相バリアントとオーバーロードは普通に分割すればよく、オブジェクトに関しては今回の多相と同様に定義を含めるよう分割する必要があると考えている。

また実装的な方向性としては、今回対象とした Caml Light より広く使われている OCaml での実装が挙げられる。多相バリアントやオブジェクトが含まれるため、拡張に関して考察する必要があるが、今回の実装とほぼ同様に実装が出来ると考えている。

さらに改良的な方向性としては、デバッグを行っている際に、プログラマの意図した型を受け取っているため、その型を使用してその型に一致するプログラムを提案することが出来ないかと考えている。これには [4] のように構文的に行う方法、またはユーザーテストなどで使用された型エラーとその修正から類似する型エラーを検出して行う方法などが考えられる。

**謝辞** 多くの有益なコメントを下さった査読者の皆様に深く感謝致します。

## 参考文献

[1] Chitil, O. “Compositional Explanation of Types and Algorithmic Debugging of Type Errors,” *Proceedings of the sixth ACM SIGPLAN international conference on Functional programming (ICFP’01)*, pp. 193–204 (2001).

[2] Haack, C., J. B. Wells. “Type Error Slicing in Implicitly Typed Higher-Order Languages,” *Science of Computer Programming - Special issue on 12th European symposium on programming (ESOP 2003)*, Volume 50 Issue 1-3 (2004).

[3] Lee, O., K. Yi. “Proofs about a Folklore let-polymorphic Type Inference Algorithm,” *ACM Transactions on Programming Languages and Systems*, pp. 707-723 (1998).

[4] Lerner, B. S., M. Flower, D. Grossman, C. Chambers. “Searching for Type-Error Messages,” *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation (PLDI’07)*, pp. 425–434 (2007).

[5] McAdam, B. J. “Generalising techniques for type debugging,” In Trends in Functional Programming, chapter 6. Intellect, (2000).

[6] Nilsson, H. *Declarative Debugging for Lazy Functional Languages*, PhD thesis, Linköping, Sweden (1998).

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

[8] Silva, J., and O. Chitil. “Combining Algorithmic Debugging and Program Slicing,” *Proceedings of the 8th ACM SIGPLAN international conference on Principles and practice of declarative programming (PPDP’06)*, pp. 157–166 (2006).

[9] Simon, A., O. Chitil and F. Huch. “Typeview: A tool for understanding type errors,” *Draft Proceedings of the 12th International Workshop on Implementation of Functional Languages,* PP. 63–69 (2000).

[10] Tsushima, K., and K. Asai. “Report on an OCaml type debugger,” *ACM SIGPLAN Workshop on ML*, 3 pages, (2011).

[11] Wand, M. “Finding the Source of Type Errors,” *Proceedings of the 13th ACM SIGACT-SIGPLAN symposium on Principles of programming languages (POPL ’86)*, pp. 38–43 (1986).