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

対馬 かなえ　　浅井 健一

お茶の水女子大学

tsushima.kanae@is.ocha.ac.jp　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器をそのまま使い，型エラースライスを作成する手法について述べる．型エラースライスとは型エラーのあるプログラムのうち，型エラーに関係がある箇所のみを抽出したプログラムである．型スライサーが実装できれば，我々の先行研究の型デバッガにスライスを導入することで動作を改良出来る．

本論文では，コンパイラの型推論器を使用して，二段階の操作で型エラースライスを求める手法を提案する．まず一段階目としては，型エラーのあるプログラムの一部を型制約を持たない式に置き換え，コンパイラの型推論器にかけて型エラーか否かを判別することで，連結した型エラースライスを求める．二段階目としては，一段階目で求めた型エラースライスに対して一部を型制約を持たないが部分項を持つような式に置き換え，一段階目と同様にコンパイラの型推論器にかけることで，より小さい型エラースライスを求める．

詳細には，let 多相を持つプログラムを対象として提案手法を説明する．型エラースライスの性質として極小性および局所性を定義し，それらを満たすスライスを求める手法について述べる．また，その手法を OCaml の型推論器を用いて実装した型スライサーと，それから得られたスライスをどのようにデバッグで使用するかについて述べる．

1　はじめに

型推論を持つ強い型付き言語には，型を書く必要がないという利点がある．その一方，正しく型がつかないプログラムを書いた場合，そのデバッグは簡単ではない．型エラーのあるプログラムに対してコンパイラはエラーメッセージを返すが，エラーメッセージでエラーの原因を特定出来るとは限らない．実際，エラーの原因がコンパイラのエラーメッセージから離れた箇所であるということもしばしば起こる．エラーメッセージが実際のエラーの原因となった箇所を特定してくれるのが，プログラマにとって最も望ましい挙動であるが，残念ながらそれは一般には不可能である．なぜなら，型エラーがあるひとつのプログラムを考えた場合でも，プログラマの意図によってエラーの原因となる箇所が異なるためである．

そこで Chitil はプログラマが意図した型を答えていくだけで，エラー箇所を特定出来るデバッガ [1] を提案した．デバッグを行う際，普通の型推論木ではエラー箇所を特定することが出来ない．そのため，Chitil は合成的に型を求めることによって，最汎型木を導出した．最汎型木とはそれぞれの式に対して最も一般的な型を持つような木である．我々の先行研究 [9, 10] では，Chitil が合成的に型推論を行うことで作成していた最汎型木を，コンパイラの型推論を使用して求めることで，デバッガ自身が型を求める必要をなくした．それにより，容易にデバッガが作成出来るようになり，OCaml のほぼフルセットを対象としたデバッガを実装することが出来た．

先行研究の問題点　先行研究 [1, 9, 10] では，作成した最汎型木に単純にアルゴリズミックデバッギング [6] を適用することで，デバッグを行っていた．アルゴリズミックデバッギングとは，一般的に木の構造を持つもののデバッグに使用される手法であり，ユーザの意図を受け取ることで，自動的に木の中を歩いてエラーの箇所を特定する．先行研究の問題点は，アルゴリズミックデバッギ

```
{}⊢^ :string -> string -> string  {x:'a}⊢x:'a  {n:'b}⊢n:'b
                {n:string, x:string}⊢x ^ n:string
List.map:省略   {n:string}⊢(fun x -> x ^ n):string -> string   lst:'c
      {n:string, lst:string list}⊢List.map (fun x -> x ^ n) lst:string list
 {n:string}⊢(fun lst -> List.map (fun x -> x ^ n) lst):string list -> string list
{}⊢(fun n lst -> List.map (fun x -> x ^ n) lst):string -> string list-> string list
             {}⊢f:string -> string list-> string list                2.0:float
                                 f 2.0 ··· Type Error
```

図 1. 最汎型木の例

ングを適用する際に木のすべてを対象としていたことに依る．その問題点を次のプログラムから考えてみる．

```
let f n lst = List.map (fun x -> x ^ n) lst in f 2.0
```

このプログラムでは，枠で囲まれた二カ所の部分の型 (の一部) が，単一化が出来ないために型エラーが起こっている．このプログラムのデバッグを行う際に用いられる最汎型木を図 1 に示した．先行研究では，デバッグを行う際にこの木全体をエラーの原因探索の対象としていた．しかしプログラムから直観的に分かるように，この木には型エラーに関係がない部分が存在する．例えば，図 1 で四角で囲まれている `List.map` や `lst`，`x` などの式は，生じた型エラーに関係がない．型エラーが起きたのは，`f` の第一引数である `n` が string → string → string 型の結合関数 `^` によって (`x ^ n` の部分で) string 型であると決められたにも関わらず，そこに float 型の `2.0` を渡しているためである．このようにエラーと無関係な部分がデバッグの対象となっていると，実際には問題になった型エラーには関係ないにも関わらず，それらに関する質問に答える必要がある．

もちろん，このようなデバッグが間違ったことを行っている訳ではない．例えば，`List.map` の型が意図とは違ったとすると，デバッグによってその食い違いを修正する箇所を見つけることが出来，その結果は正しい．しかし，そのような部分をどう修正したとしても，今考えている `^` と `2.0` の型の衝突を直すことはできない．また，実際に起きた型エラーと無関係である箇所に別のエラーがあるとは限らない．無関係な箇所を聞く挙動は，ユーザの負担を増やすだけで出来る限り避けるべきである．特にデバッグ対象のプログラムが大きくなればなるほど，発生した型エラーとは無関係な箇所を聞くことの負担は増し，実用上問題となってくる．

解決法：スライスの使用　そこで我々は最汎型木に型エラースライスを用いて，デバッグの範囲を狭めることを考えた．プログラムのスライスとは，あるプログラムのうち特定のことに関係する部分のみを集めたものである．何に関してスライスを取るかによって，いくつかの種類に分けることが出来るが，その中でもプログラムのエラーに関係するものでは，型エラー [2, 8]，意味的エラー [7] などに対してスライスが提案されている．エラーに関するスライスの利点は，プログラマの入力なしにエラーに関係する範囲を特定することが出来，デバッグなどの際に見る範囲を狭めることが出来ることである．型エラースライスでは，型エラーのあるプログラムのうち，型エラーに関係した部分を抜き出したプログラムになる．例えば，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 f n ... = ... (... ^ n) ... in f 2.0
```

上のスライスの `...` 以外の部分がスライスに含まれる部分である．このようなスライスを用いることで，デバッグする範囲を狭めることが出来，実際にエラーに関係する部分のみを見られるようになる．

既存研究での型スライスの作成法　一般的に型エラースライスは，型エラーを起こす最小の組み合わせを求めることで得られる．既存研究 [2] では，型エラーのあるプログラムのスライスを求める際，型規則を用いてそのプログラムによって必要とされる型制約を導き，その中から型エラーを起こす最小の組み合わせを求めることで型エラースライスを求めていた．この手法ではコンパイラの型推論器とは別に，型スライサーが型規則を持つ必要がある．型スライサーが型規則を持つことの欠点はふたつ挙げられる．まずひとつめとしては，型スライサーの持つ型規則がコンパイラの型推論が用いている型規則と異なっている場合，正しい型スライスを求めることが出来ないという点である．本来エラーに関係がある部分がスライスに含まれていない場合，デバッグを行っても正しい箇所を特定することが出来ない．ふたつめとしては，実装が大変である点である．型スライサーには型推論を行う必要はないが，型エラーか否かを調べるために，型推論器と同様に構文や型の定義・単一化などを必要とする．

型制約を用いた手法では，汎用的に用いられている強い型付き言語のフルセットに対する型スライサーとして，[2] を拡張して実装された SML を対象言語とした Skalpel [1] のみが実装されている．実装上の労力が要因なのか，他の言語では実装が行われていない．我々のデバッガ [9, 10] に使用するためには OCaml フルセットでの実装が必要になるため，本研究では異なる手法を用いる必要があった．そこで我々はコンパイラの型推論器を用いて，デバッグに適した型エラースライスを求めることを目的とした．[2]

本研究の目的　本研究は，コンパイラの型推論器を用いて型エラーのスライスを求めることを目的とする．既存手法 [2, 8] でスライスを求める時に使用される型規則は，それらの単一化を行った場合に型エラーになるか否かという判断に使われている．その判断をコンパイラの型推論に委ね，あるプログラムが正しく型がつくか・つかないかという結果を受け取ることで，スライスを求める．

本論文の流れは次のようになる．まず，次節で型エラースライスとその性質に関する定義を行い，その後本論文で型スライスを作成するためのアイデアを述べる．本論文では対象言語を let 多相を含むプログラムとして，それに対するスライスを求めていく．まず一段階目として連結なスライスを (3 節)，二段階目として非連結なスライスを求める (4 節)．これらによってデバッグに使用できる型エラースライスを求めることが出来る．その後，それらが 2 節で定義した性質を満たしているかどうかの議論を行う (5 節)．そして，OCaml の型推論器を用いた型スライサーの実装とスライスをデバッグでどのように使用するかを述べ (6 節)，既存の型スライス作成手法や型デバッグ手法などの関連研究との比較を行う (7 節)．最後に本論文をまとめ，今後の課題について考察する (8 節)．

2　型エラースライスと提案手法

本節では，型エラースライスとは何かを例を交えて説明し，型エラースライスとその性質について定義を行う．その後，型エラースライスを求めるために本論文で用いるアイデアを簡単に説明する．

型エラースライス　前節で触れた通り，型エラースライスとは型エラーに関係する部分のみを集めたプログラムである．例として，`(fun x -> (fun y -> x + 1)) true` というプログラムを考えてみる．このプログラムでは `+` と `true` というふたつの部分によって型エラーが生じている．具体的には `+:`[`int`]`->int->int` と `true:`[`bool`] の枠で囲まれたふたつの型を単一化出来なかったことがエラーの原因である．このようなふたつの型を特定するのは可能であるが，常にふたつの部分に

[1] http://www.macs.hw.ac.uk/ultra/skalpel/index.html
[2] 我々と同様のアイデア，コンパイラの型推論器を用いて型エラースライスを作成する手法 [5] との比較は関連研究にて行う．

$M : term$	::=	`c`	(定数)
	|	`x`	(変数)
	|	`fun x ->` M	(ラムダ抽象)
	|	M `@` M	(関数適用)
	|	`let x =` M `in` M	(変数定義)
$\tau : type$	::=	b	(型変数)
	|	$\mathrm{int}, \mathrm{bool}, \ldots$	(型定数)
	|	$\tau \to \tau$	(関数型)

図 2. 対象言語の構文と型

$S : slice$	::=	`c`
	|	`x`
	|	`fun x ->` S
	|	S `@` S
	|	`let x =` S `in` S
	|	$\square$
	|	$S \mathbin{\texttt{@}\!\!\!\!=} S$

図 3. スライスの構文

エラーの原因があるとは限らない．(例えば，上の例ではユーザは `x + 1` ではなく `y + 1` と書くつもりだったのかもしれない．) 型エラーのスライスとは，このような衝突する型をもつふたつの部分と，それらが単一化されなくてはならないという制約を導入した部分を集めたプログラムである．前述の例では，単一化出来ないふたつの型を導入した `+` と `true`，そしてそれらふたつの型が単一化されなくてはならないという制約を導入した関数適用とラムダ抽象が型エラーのスライスに含まれる．よって，この例に対するスライスはそれらの集合 `(fun x -> (... x + ...)) true` になる．

複数の型エラースライス　型エラーのあるひとつのプログラムに対して，複数のスライスが存在することがある．例えば `(true + false)` というプログラムを考えてみると，`(true + ...)` と `(... + false)` というふたつのスライスが考えられる．これは前述した型の衝突がひとつだけではなく，複数存在したためである．特にプログラムが大きくなると，型の衝突の数が増え，それに従ってスライスの数も多くなることが多い．すべての衝突の可能性を洗い出すのは容易ではなく，既存手法 [2] でも計算量の問題から，すべて求めてはいない．

デバッグの観点からすれば，型エラースライスがひとつ求まればそれに従ってデバッグすることが出来る．よって複数のスライスが考えられる場合にすべて求めることはそこまで重要ではないという考え方をすることも出来る．しかしふたつの可能性が考えられる．ひとつは，デバッグしやすいスライスとしにくいスライスが存在する可能性である．単純な例では，離散して現れるスライスよりもまとまって現れるスライスの方がデバッグがしやすいと考えられる．もうひとつの可能性は，複数のスライスを用いることで，デバッグが容易になることである．前述のプログラムで `(true + ...)` と `(... + false)` という複数のスライスを考えた時，これらの共通部分は `+` である．このようにある部分が複数のスライスに含まれる場合には，エラーの可能性が高い．複数のスライスからエラーの可能性が高いところを検出し，デバッグの際にそのような場所から質問することが出来れば，デバッガの利便性を高めることが出来る．よって本研究では，スライスに対してひとつの基準を提案し，それを満たすような型エラースライスをできるだけ多く求めることを目的とした．

連結でない型エラースライス　型エラーのスライスが常に全ての部分が構文木上で繋がった連結なプログラムであるとは限らない．型推論では式の型だけでなく，型環境も型エラーに影響する．それによってスライスの一部が離れた箇所に存在し，連結でないスライスになることがあり得る．例えば，`(fun x -> ((x + 1) + 2, x 3))` というプログラムでは，`(fun x -> (... (x + ...) ..., x 3))` というスライスが考えられる．このスライスは `... + 2` という関数適用部分がスライスに含まれないことを示している．

言語　型エラースライスの定義を行う前に，本論文で対象とする構文と型の定義 (図 2) について述べる．構文には，定数，変数，ラムダ抽象，関数適用などのラムダ計算に加え，let 多相を持つ変数

定義が含まれる．+ などのオペレータは初期環境に含まれており，それらは全て定数 c に含まれると仮定する．

本研究ではコンパイラの型推論器を用い，プログラムが型エラーになるか否かだけを扱うため，直接には型を扱わないが，想定した型は図 2 に示した型変数，型定数，関数型である．型推論後の型のみを考えているため，型スキームは含まない．**infer_type** はコンパイラの型推論器で，式 M を与えると型 τ を返し，型エラーの際には **TYPE_ERROR** という例外を返すものと仮定する．本研究では **infer_type** が返す型は使用せず，型エラーが起きるかどうかの情報のみを使用する．

型エラースライスの定義　型エラースライスの構文は，図 3 のように定義出来る．これは対象言語の構文に，□ と S @ S という特殊構文を追加したものである．□ は型の制約を持たず，どんな型とも単一化出来る型を持つ式である．コンパイラの型推論を使用する際，□ は例外などで表現することが出来る．S_1 @ S_2 は，S_1 と S_2 のそれぞれがお互いに型の制約を持たず，プログラム S_1 @ S_2 としてどんな型とも単一化出来る型を返すが，部分項の型環境のみが共有される式である．こちらは，例外を関数として S_1 と S_2 を引数として渡すことで表現することが出来る．

定義 1 (型エラースライス) スライス S をコンパイラの型推論に渡した結果が型エラーになる，つまり以下が成り立つならば，スライス S は型エラースライスであるという．

infer_type S = **TYPE_ERROR**

これは一見当たり前のようであるが，対象言語の定義がスライスの定義に包含されていることから分かるように，この条件は型エラーになる元のプログラムも一種の型エラースライスであることを示している．

よって，スライスにおいては極小性が重要である．スライスの極小性は，次に示すスライス間の包含関係を用いて定義できる．

定義 2 (スライス間の包含関係) スライス S の一部を □ や @ で置き換えたスライス $S'(S \neq S')$ を考えたとき，スライス S とスライス S' には包含関係が存在し，$S \supset S'$ と表す．

定義 3 (極小性 (Minimality)) 型エラースライス S に対して $S \supset S'$ となるスライス S' を考える．どのような S' も必ず型が付くとき，つまり以下が成り立つならば S を極小な型エラースライスであるという．

infer_type S = **TYPE_ERROR** $\wedge$ $\forall S', S \supset S'$, **infer_type** S' = τ

この条件によって極小なスライスの全ての部分が型エラーに関係があり，余計なものが含まれていないことが保証される．Haack と Wells はスライスの極小性を型規則の集合によって定めた [2] が，我々は構文によって定めていると考えることが出来る．

極小なスライスは必要最低限の箇所しか含まないため，デバッグする上で有用である．しかし，ある型エラーのプログラムに関する極小なスライスをすべて求めることは，前述したとおり計算量の観点から容易ではない．よって本研究では，スライスに関する性質として局所性を提案し，最も局所性の高いスライスを求める．

局所性を説明するための例として，単純なプログラム `let v = ((1 + 2.) +. 3) in v + 4.` を考える．このプログラムでは型の衝突が多数存在している．そのため，極小なスライスは 5 つ考えられる．図 4 にこれら 5 つの極小なスライスの木を示した．局所性の直観的な説明としては，出来るだけスライスに含まれる要素が近くに存在することである．図 4 ではスライスがどのくらい広がっているか (領域) を四角で囲って示した．

局所性には，スライス内の要素が関わってくる．ある要素について考えたとき，あるスライス S よりもより近くに収まっているスライス S' があるならば，そちらの方が局所性が高いといえる．図

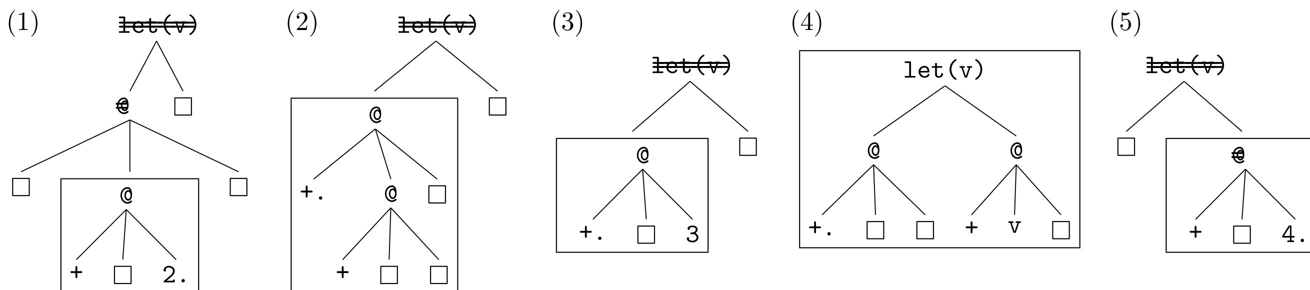


図 4. `let v = ((1 + 2.) +. 3) in v + 4.` の極小なスライス

4 の木で考えてみると，(3) と (4) は共通の要素がいくつか存在し，(3) より (4) の方が領域が広い．よって (3) の方が局所的であるといえる．(1) と (2) も同様に，共通する要素が存在し，(1) の方が領域が狭い．よって (1) の方が局所的である．(1) と (3) ではそもそも共通する要素が存在せず，(5) も同様である．これら 5 つのスライスから最も局所性の高いスライスを選ぶと，(1)，(3)，(5) の 3 つが得られる．このような局所性を定義するために，まずスライス間の共有とスライスの領域というふたつの概念を定義する．

定義 4 (スライス間の要素の共有) 型エラーのあるプログラム M の型エラースライス S で，スライス S の一部が型エラースライス S' に含まれるとき，S と S' にはスライス間で要素の共有が存在するという．

定義 5 (スライスの領域) 型エラーのあるプログラム M の型エラースライス S を考え，スライス S で □ や @ ではない最も上のノードを考える．その頂点を根とする M の部分木をスライスの領域と呼び，$region_M(S)$ と表す．(図 4 の四角に相当する)

定義 6 (局所性 (Locality)) 同じ型エラープログラム M の型エラースライス S と型エラースライス S' でスライスの要素が共有しており，スライス S の領域にスライス S' の領域が包含されるとき ($region_M(S) \supset region_M(S')$)，スライス S よりもスライス S' の方が局所性が高いという．また，より局所性のあるスライスが存在しないスライスを，最も局所性の高いスライスという．

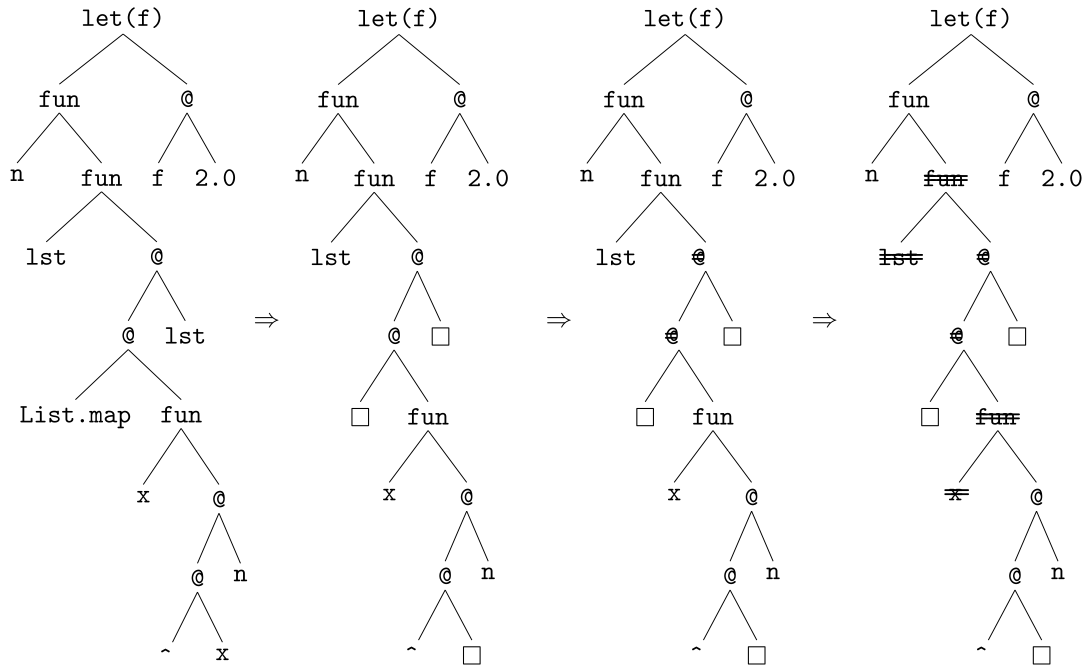
提案手法　極小なスライスを求めるには，型エラーのあるプログラム M の部分プログラムを，スライスで新たに導入した □ や S @ S で出来るだけ置き換えればよい．置き換え後のプログラムでも型エラーが起こるならば，置き換えられた部分は型エラーに関係がないと判断することが出来る．本論文では二段階の操作で置き換えを行う．一段階目はプログラムの部分木の □ への置き換えである．この操作では部分木の置き換えのみなので，得られるスライスは全ての部分が連結している．二段階目はスライスの中間ノードの S @ S への置き換えである．S @ S で置き換えられた中間ノード自体はスライスには含まれないため，ここで得られるスライスは非連結なものになる (置き換えられる中間ノードが存在しない場合には連結なままとなる)．

前節のプログラム `let f n lst = List.map (fun x -> x ^ n) lst in f 2.0` を例として考えてみる．このプログラムを木にしたものが，次頁に示した図の一番左の木である．まず全体が型エラーになるように保ちながら，なるべく多くの部分木を □ で置き換えると，

```
let f n lst = □ (fun x -> □ ^ n) □ in f 2.0
```

という連結なスライス (左から二番目) が得られる．このスライスの全体が型エラーになるように保ちながら，なるべく多くの束縛ではない中間ノードを S @ S で置き換えると，

```
let f n lst = (□ @ (fun x -> □ ^ n)) @ □ in f 2.0
```



という非連結なスライス (右から二番目) が得られる．@ のノードはスライスに含まないとすると，この時点で 3 つの部分に分かれたスライスである．最後に後処理として，スライスに不要な束縛を除去すると，

```
let f n ...  = ... (... ^ n) ... in f 2.0
```

というスライス (一番右) が得られる．このような操作を順に行うことで，デバッグで使用するスライスが求められる．具体的には，3 節で連結なスライスの集合を求め，4 節で連結なスライスから非連結なスライスを求める．

3　連結な型スライスを求める

本節では，最初の段階として連結な型エラースライスを求める．この変換が本論文での中心的な変換である．具体的にはプログラム中の各部分木を型の制約をもたない式 □ で置き換え，型エラーになるかどうかを観測する．置き換えたプログラムが型エラーになるならばそれは型エラースライスであり，□ で置き換えられなかった部分項を □ で置き換えることで，より小さいスライスを求めることができる．

例　`(fun x -> x + x) (3.0 +. 2.0)` というプログラムを例として，連結な型スライスを求めることを考える．まず，一番外側の関数適用の部分項を □ で置き換えることを考える．関数部分と引数部分，それぞれを □ で置き換えるかどうかで 4 つの候補が考えられるが，そのうち型エラーになるのは `(fun x -> x + x) (3.0 +. 2.0)` だけである．これもひとつのスライスであるが，部分項をより小さいスライスに置き換えたい．そこで次は部分項のひとつの `(fun x -> x + x)` を対象として，そのスライスを求める．

ここで，`(fun x -> x + x)` というプログラム単体には型がつくことから分かるように，型エラーになるかどうかを観測したいのは対象としたプログラムの全体である．よって，置き換えを行うためには，対象となるプログラムだけでなくそのコンテキストが必要になる．`(fun x -> x + x)` のコンテキストは，そのまわりのプログラム `(fun k -> k (3.0 +. 2.0))` となる．このようなコンテキストへ対象となっているプログラムを適用することで，全体のプログラムが得られる．この全

体のプログラムが型エラーになるように保ちながら，□ への置き換えを行えば良い．このとき，コンテキストは単独で型がつくという制限を加える．詳しくは後述するが，この不変条件を満たしながら置き換えを行うことで，最も局所的で極小なスライスを得ることが出来る．

対象プログラム（`fun x -> x + x`）のスライスをコンテキスト（`fun k -> k (3.0 +. 2.0)`）の元で考える．ここで，`fun` 文の中身の関数適用について考えてみる．この関数適用の要素は 3 つなので，普通にすべての場合を考えると $2 \hat{} 3 = 8$ 通り考えられる．そのうちコンテキスト（`fun k -> k (3.0 +. 2.0)`）のもとで型エラーになるのは（`fun x -> x + x`）と（`fun x -> x + □`），（`fun x -> □ + x`）の 3 つなので，これらが候補として返ってきそうである．しかし，先ほどの不変条件のおかげで，これら 3 つのうち，ふたつの極小なスライス（`fun x -> x + □`）と（`fun x -> □ + x`）のみが得られる．これは極小でないスライス（`fun x -> x + x`）では，そのスライスの中でより置き換えを行おうとしたとき，コンテキストの型がつかないことによる．例えばひとつめの `x` を考えると，そのときのコンテキストは（`fun k -> (fun x -> k + x) (3.0 +. 2.0)`）であり，型がつかない．このような不変条件を守りながら置き換えを行うことで，関数部分のスライス（`fun x -> x + □`）と（`fun x -> □ + x`）が得られる．

関数部分のスライスが求まったため，引数部分に対象を移して考えてみる．まず関数部分のスライスのうちのひとつ，（`fun x -> x + □`）に関して考える．引数部分（`3.0 +. 2.0`）のスライスを求める際のコンテキストは，（`fun k -> (fun x -> x + □) k`）となる．このコンテキスト上でスライスを考えると，（`3.0 +. 2.0`），（`3.0 +. □`），（`□ +. 2.0`），（`□ +. □`）という 4 つの候補が考えられそうである．しかし関数部分と同様にこの中で極小なスライス（`□ +. □`）だけが得られる．

これらの操作により，このプログラム全体の型エラースライスのひとつは（`fun x -> x + □`）（`□ +. □`）であると求められた．同様にもうひとつの関数部分のスライス（`fun x -> □ + x`）に対しても，極小なスライスを求めると，（`fun x -> □ + x`）（`□ +. □`）という型エラースライス求まり，これらふたつが元のプログラムのスライスとして得られる．

プログラム　このような操作を行う関数を図 5 に示した．*cut* はプログラム M とコンテキスト cxt を受け取り，そのコンテキストの元での M のスライスのリストを返す関数である．*cut* ではコンテキスト cxt には常に型がつき，そのコンテキスト cxt とプログラム M を合わせたもの $cxt(M)$ には常に型がつかないという不変条件が満たされている．

変数や定数の場合には，それ自身だけがスライスとして返される．これは cxt と合わせた場合に型エラーになる M しか，*cut* に渡されないことによる．ラムダ抽象 `fun x ->` M の場合は，部分項がひとつなので，その部分項 M をより小さいスライスにするべく再帰を行えばよい．再帰によって M のスライスのリストが得られるので，それらに `fun x ->` という構文をかぶせて返している．

関数適用と変数定義は，部分項が複数あるので少し複雑になる．*cut* の関数適用では，それぞれの部分項をコンテキストに入れた際に型がつくかどうかを観測している．その結果によって，不変条件を守るために，以下のように分類される．

$(\lambda k.cxt(M_1 @ k))$	$(\lambda k.cxt(k @ M_2))$	候補となるスライス
τ	τ	$M_1 @ M_2$
TYPE_ERROR	τ	$M_1 @ \square$
τ	**TYPE_ERROR**	$\square @ M_2$
TYPE_ERROR	**TYPE_ERROR**	$(M_1 @ \square)$ と $(\square @ M_2)$

両方に型がつく場合には，M_1 と M_2 を合わせた場合のみに型エラーがおこるということなので，候補となるスライスは $M_1 @ M_2$ である．どちらかに型がつかない場合，$M_1 @ M_2$ を候補としてしまうと，部分項に再帰した場合に，コンテキストに型がつかない．よって型エラーとなった片方の部

$$
\begin{array}{rcl}
cut : slice * (slice \rightarrow slice) & \rightarrow & slice\ list \\
cut[\![(\mathtt{c}, cxt)]\!] & = & [\mathtt{c}] \\
cut[\![(\mathtt{x}, cxt)]\!] & = & [\mathtt{x}] \\
cut[\![(\mathtt{fun\ x\ \text{->}}\ M', cxt)]\!] & = & let\ lst = cut[\![(M', (\lambda k.cxt(\mathtt{fun\ x\ \text{->}}\ k)))]\!]\ in \\
 & & map\ (\lambda y.(\mathtt{fun\ x\ \text{->}}\ y))\ lst \\
cut[\![(M_1\mathtt{@}M_2, cxt)]\!] & = & try(\mathbf{infer_type}(\lambda k.cxt(M_1\mathtt{@}k)); \\
 & & \quad (try(\mathbf{infer_type}(\lambda k.cxt(k\mathtt{@}M_2)); \\
 & & \qquad cut'[\![(M_1\mathtt{@}M_2, cxt)]\!]) \\
 & & \qquad with\ \mathbf{TYPE_ERROR} \rightarrow cut'[\![(\Box\mathtt{@}M_2, cxt)]\!]) \\
 & & with\ \mathbf{TYPE_ERROR} \rightarrow \\
 & & \quad try(\mathbf{infer_type}(\lambda k.cxt(k\mathtt{@}M_2)); \\
 & & \qquad cut'[\![(M_1\mathtt{@}\Box, cxt)]\!]) \\
 & & \quad with\ cut'[\![(M_1\mathtt{@}\Box, cxt)]\!] \cup cut'[\![(\Box\mathtt{@}M_2, cxt)]\!] \\
cut[\![(\mathtt{let\ x\ =}\ M_1\ \mathtt{in}\ M_2, cxt)]\!] & = & \text{省略 (関数適用と同様)} \\
 & & \\
cut'[\![(M_1\mathtt{@}M_2, cxt)]\!] & = & let\ M_1's = if\ M_1 = \Box\ then\ [\Box] \\
 & & \qquad\qquad else\ cut[\![(M_1, (\lambda k.cxt(k\mathtt{@}M_2)))]\!]\ in \\
 & & let\ lst = map(\lambda M_1'.(M_1', \\
 & & if\ M_2 = \Box\ then\ [\Box]\ else\ cut[\![(M_2, (\lambda k.cxt(M_1'\mathtt{@}k)))]\!]))M_1's\ in \\
 & & flatten\ (map\ (\lambda(M_1', M_2's).(map\ (\lambda M_2'.(M_1'\mathtt{@}M_2'))\ M_2's))\ lst) \\
cut'[\![(\mathtt{let\ x\ =}\ M_1\ \mathtt{in}\ M_2, cxt)]\!] & = & \text{省略 (関数適用と同様)}
\end{array}
$$

図 5. 連結な型スライスへの変換 *cut* とその補助関数 cut'

分項のみをもつようなスライスを候補とする．両方が型エラーであった場合には，それらの和集合となる．この操作と不変条件件によって，極小性と局所性が保たれている．

補助関数 cut' は，*cut* と同様にプログラム M とコンテキスト cxt を受け取り，プログラム M の型エラースライスのリストを返す．cut' では，不変条件に加えて，部分項に対してある性質が満たされている．cut' の受け取るプログラム M の部分項が □ 以外のスライスならば，コンテキスト cxt に部分項を埋め込んだコンテキストにも型がつく．cut' の関数適用をみてみると，まず部分項 M_1 からより小さいスライスのリスト $M_1's$ を得る．そして $M_1's$ のひとつひとつの要素をコンテキストとして埋め込んだ新しいコンテキストのもとで M_2 をより小さいスライスにする．このようにして得られたスライスを，関数適用の構文に埋め込んだものが関数適用の型エラースライスとなる．

変数定義では多相型が導入されるため，多少複雑になりそうである．しかし，本手法ではコンパイラの型推論を用いているので，単純にコンパイラが返してきた型を使うだけで良い．関数適用と変数定義の規則はコンテキストに使われる構文や最後に埋め込む構文が違うだけであるため，同じような操作で実現される．if 文などに関しても，同様に不変条件を守りながら部分項に対して段階的にスライスを求めていくことで変換を行うことができる．

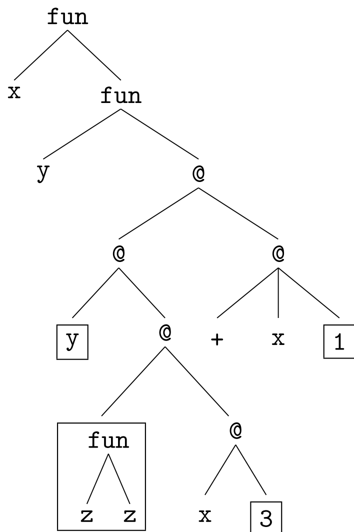
4 非連結な型スライスを求める

本節では，二段階目として 3 節で求めた連結なスライスを元にして，非連結な型スライスを求める．非連結な型エラースライスを求めるには，スライスに不要な中間ノードを取り除けばよい．

例として `(fun x -> fun y -> y ((fun z -> z) (x 3)) (x + 1))` というプログラムを考える．このプログラムを木にすると，以下のようになる．このプログラムは，`(x 3)` で `x` を関数と

$$
\begin{array}{rcl}
\mathit{remove_node} : \mathit{slice} * (\mathit{slice} \to \mathit{slice}) & \to & \mathit{slice} \\
\mathit{remove_node}[\![(\mathtt{c}, \mathit{cxt})]\!] & = & c \\
\mathit{remove_node}[\![(\mathtt{x}, \mathit{cxt})]\!] & = & x \\
\mathit{remove_node}[\![(\square, \mathit{cxt})]\!] & = & \square \\
\mathit{remove_node}[\![(\mathtt{fun\ x\ ->}\ M, \mathit{cxt})]\!] & = & \mathtt{fun\ x\ ->} \mathit{remove_node}[\![(M, (\lambda k.\mathit{cxt}(\mathtt{fun\ x\ ->}\ k)))]\!] \\
\mathit{remove_node}[\![(M_1 \mathtt{@} M_2, \mathit{cxt})]\!] & = & \mathit{try}(\mathbf{infer_type}(\mathit{cxt}(M_1 @\ M_2)); \\
 & & \quad ((\mathit{remove_node}[\![(M_1, (\lambda k.\mathit{cxt}(k \mathtt{@} M_2)))]\!]) \mathtt{@} \\
 & & \quad\ (\mathit{remove_node}[\![(M_2, (\lambda k.\mathit{cxt}(M_1 \mathtt{@} k)))]\!])) \\
 & & \mathit{with}\ \mathbf{TYPE_ERROR} \to \\
 & & ((\mathit{remove_node}[\![(M_1, (\lambda k.\mathit{cxt}(k \mathtt{@} M_2)))]\!]) @ \\
 & & \ (\mathit{remove_node}[\![(M_2, (\lambda k.\mathit{cxt}(M_1 \mathtt{@} k)))]\!])) \\
\mathit{remove_node}[\![(\mathtt{let\ x=}M_1\ \mathtt{in}\ M_2, \mathit{cxt})]\!] & = & \mathit{let}\ x = \mathit{remove_node}[\![(M_1, (\lambda k.\mathit{cxt}(\mathtt{let\ x=}k\ \mathtt{in}\ M_2)))]\!]\ \mathit{in} \\
 & & \mathit{remove_node}[\![(M_2, (\lambda k.\mathit{cxt}(\mathtt{let\ x=}M_1\ \mathtt{in}\ k))]\!]
\end{array}
$$

図 6. 不要な中間ノードの除去 *remove_node*



して使っているにも関わらず，`(x + 1)` では x を int として使っていることから型エラーが生じている．このプログラムで連結な型エラースライスを求めると，`(fun x -> fun y -> □ (□ (x □)) (x + □))` というスライスが得られる．左の木で四角で囲まれている部分が，3 節で □ に置き換えられた部分であり，囲まれていない部分が型エラースライスとして残っている部分である．このプログラムを具体例として，不要な中間ノードの除去を考える．

この例の中間ノードのうち，上から三段目の @，四段目左ノードの @，五段目の @ に示されている関数適用は実は不要なノードである．これらは関数適用の制約によって型スライスに含まれるのではなく，型環境が同じでなければならないという制約によって型スライスに含まれている．具体的には `x 3` と `x + 1` がプログラムに含まれているため，型環境で単一化出来ないという状況になっている．これらが関数適用の制約と無関係であることは，その規則を関数適用でなくペアにしたとしても同じ結果の型エラーになることから分かる．このようにそれ自身の規則に関わらずスライスに含まれているノードを取り除くのが，本節での変換である．

例　このようなノードを除くには，そのノードを制約を持たない構文 S @ S で置き換え，型エラーになるかどうかをみればよい．例えば，上から三段目の @ の置き換えを考えてみる．置き換えると，`(fun x -> fun y -> (□ (□ (x □))) @ (x + □))` というスライスが得られるが，このスライスにも型がつかない．このことから，型エラースライスにとって，この部分の構文が関数適用である必要がないことが分かる．よって，より小さいスライスになるように @ に置き換える．

これは関数適用に限らず if 文など複数の部分項を持つものは同じように変換が出来る．その式自体の制約が無関係である場合には @ で置き換えることによって，より小さいスライスにすることが出来る．

プログラム　図 6 に不要な中間ノードを削除する関数 *remove_node* を示した．*remove_node* はスライス M とそのコンテキスト *cxt* を受け取り，不要なノードを @ で置き換えたスライスを返す関数である．定数・変数・□ は中間ノードではなく葉であり，型エラースライスに不必要ならば 3 節の時点で削除されているので，ここではそのまま残す．ラムダ抽象と変数定義はそれぞれ部分項に対して再帰を行っている．関数適用 M_1 @ M_2 では，まず関数適用を M_1 @ M_2 に置き換えたときに型エラーになるかどうかを観測している．M_1 @ M_2 が型エラーになるならばその関数適用はスライスに不要なので取り除き M_1 @ M_2 として，型エラーにならないならばその関数適用はスライスに必要なので残している．

5　正当性に関する考察

本節では本論文で示した変換によって得られるスライスが，2 節で示した性質を満たしているかどうかの考察を行う．ここでは，*cut* に関して健全性・極小性を示す．

命題 1 (*cut* の健全性) *cut* はスライス S とコンテキスト C ($C[S]$ は型エラー, C は単独で型がつく) を受け取り，コンテキスト C の元で型エラーを起こすスライスのリストを返す．

これはスライス S に関する帰納法で示すことが出来る．同時に条件を満たすコンテキストとスライスを与えると，少なくともひとつの型エラースライスが得られることも示せる．

命題 2 (*cut* の極小性) *cut* はスライス S とコンテキスト C ($C[S]$ は型エラー, C は単独で型がつく) を受け取り，コンテキスト C の元で極小なスライスを返す．

これもスライス S に関する帰納法で示すことが出来る．□ の置き換えで分岐する可能性があるのは関数適用 M_1@M_2 の場合である．M_1 か M_2 のどちらかが単独で型エラーになる場合には自明で成り立つ．よって両方が型エラーでない場合を考える．帰納法の仮定から，コンテキストを (`fun k -> k @` M_2) としたスライス M_1' が得られる．同様に M_1' を使用して，コンテキスト (`fun k ->` M_1' `@ k`) のもとでのスライス M_2' が得られる．M_1' が (`fun k -> k @` M_2') のもとでも極小であることは，次の補題から示される．

補題 1 あるコンテキスト C の元で M が極小の型エラースライスならば，$C \supset C'$ となるコンテキスト C' (ただし $C'[M]$) においても M は極小な型エラースライスである．

これは背理法を用いて証明できる．あるコンテキスト C と C のもとで $C[M]$ が型エラーとなる極小なスライス M をとってくる．$C \supset C'$ となるコンテキスト C' において，$M \supset M'$ となるスライス M' が存在し，$C'[M']$ が型エラーになると仮定する．すると，$C[M']$ も型エラーになるはずであり，$M \supset M'$ より，コンテキスト C のもとで極小なスライスは M ではなく M' であり，矛盾が導かれる．よって，題意は満たされる．

6　実装とデバッグへの適用

本節では型スライサーの実装と，それによって得られたスライスをデバッガでどのように用いるかについて述べる．また，1 節で先行研究 [1, 9, 10] の問題点を説明するために用いた例が，スライスを使用したデバッガではどのように動くかについて述べる．

```
{}⊢^ :string -> string -> string  {n:'b}⊢n:'b
        {n:string}⊢ ...  ^ n:string
  {}⊢(fun n -> ...  (...  ^ n)):string -> 'a
              {}⊢f:string -> 'a              2.0:float
                          f 2.0 ··· Type Error
```

図 7. スライスを利用した最汎型木の例

実装 型スライサーの実装は OCaml 3.12.1 の型推論器を使用して行った．現段階では，OCaml の一部の構文だけを対象としている．型推論器は完全にブラックボックスとして扱い，スライスを渡した際に型がつくか，つかないかだけの情報を利用している．型推論器だけでなく，構文木や字句解析器・構文解析器も OCaml のものを利用した．スライスの構文で用いていた □ は `assert false`，S_1 @ S_2 は `(assert false)` S_1 S_2 と表現することで実装を行っている．

デバッグへの適用 本論文の変換で作成したスライスを，デバッグに使用する方法について述べる．元の型エラープログラムによっては複数のスライスが得られるが，デバッグを行うのはそのうちひとつのスライスで良い．これはスライスにはエラーの原因が必ず含まれていることによる．ひとつのスライスに対してデバッグを行い，エラーを特定して修正してから，再び型エラーになるようであれば再びデバッグを行うことで，複数の型の衝突は修正できる．これでは複数のスライスを得た意味がないが，複数のスライスの中からよりデバッグしやすいスライスを選んだりすることが可能である．また，発展的にはスライスを重ね合わせ，エラーの可能性が高い部分を優先的に質問する方法を検討中である．

次にスライスで新しく導入した構文 □ と S_1 @ S_2 を含むスライスをどのようにデバッグすれば良いかについて考える．□ は型の制約を持たないため，もともと存在しないものとして扱えば良い．S_1 @ S_2 は少しだけ複雑になる．基本的にはそのノードは無視して，その部分木を見に行けば良い．しかしアルゴリズミックデバッギングは完全に木の構造だけによって実装されるため，構文によってとばす等の操作は苦手である．そのため，スライスを元に最汎型木を作る際に，S_1 @ S_2 のノードをとばした木を作ることで対応できる．例として，1 節で問題点を示すために挙げたプログラム `let f n lst = List.map (fun x -> x ^ n) lst in f 2.0` に対して，スライスを使用した最汎型木を図 7 に示した．一例としては，もともとの木では `(fun n -> ...)` の上に `(fun lst -> ...)` というノードが存在したが，変換によって @ となったため，この木ではとばされている．1 節の単純に作られた最汎型木 (図 1) よりも，この最汎型木の方がデバッグの対象となる場所がしぼられていることが分かる．

実行例 実行例として，1 節で示したプログラム

```
let f n lst = List.map (fun x -> x ^ n) lst in f 2.0
```

を考えてみる．このプログラムからスライスを作成し，不要な束縛を _ で置き換えると，

```
let f = (fun n -> (fun _ -> ((assert false) (fun _ -> (assert false) ^ n)
                                    (assert false)))) in f 2.0
```

という結果が得られた．これをこれまで使用していた記法で書き直すと，以下のようになる [3]．

```
let f = (fun n -> (fun _ -> ((□ @ (fun _ -> □ ^ n)) @ □))) in f 2.0
```

ここで，プログラマの意図では，関数 `f` は `float -> float list -> float list` 型であると仮定する．スライスを使用せずにもとのプログラムをデバッグすると，

---

[3] OCaml で実装するにあたって □ を `assert false` に，S_1@ S_2 を `(assert false)` S_1 S_2 と表現したため，□ @ S_1 の場合には再変換は省略し，`((assert false) (assert false)` S_1`)` ではなく `((assert false)` S_1`)` となるように実装している．

- f 2.0 の f の型が string -> string list -> string list か? (No と答える)
- 関数 f の本体部分で n が string として使われているか? (No と答える)
- 関数 f の本体部分で lst が string として使われているか? (No と答える)
- List.map の型が ('a -> 'b) -> 'a list -> 'b list か? (Yes と答える)
- x ^ n で x が string か?(No と答える)
- ^ の型が string -> string -> string か? (No と答える)

という 6 つの質問によって，^ がエラーの原因であると特定される．同じプログラムに型エラースライスを適用してデバッグを行うと，

- f 2.0 の f の型が string -> 'a か? (No と答える)
- □ ^ n で n が string として使われているか? (No と答える)
- ^ の型が string -> string -> string か? (No と答える)

という 3 つの質問により，スライスを適用しない場合と同様に ^ がエラーの原因であると特定される．型エラースライスを使用したことにより，実際に起きた型エラーに関係する部分だけが質問されていることが分かる．具体的には，スライスに含まれない List.map や lst，x に関する質問がなくなっている．

7 関連研究

本節では，関連研究との比較を行う．型エラースライスに関するもの，デバッグに関するもののそれぞれを順にみていく．

型エラーのスライス Haack と Wells は型制約を用いて型エラースライスを作成する方法 [2] を提案した．彼らがスライスを求める目的は我々と同じデバッグであるが，より狭い範囲で考えると異なっている．我々はデバッガへの適用を目的としているが，彼らの目的はスライスをエラーメッセージとしてユーザに提供することであった．型エラースライスを用いることで，プログラムのエラーに関係する可能性をもつ範囲を縮め，ユーザが型エラーの原因を見つけやすくすることができる．彼らは型に注目し，型規則を用いて型スライスを作成した．そのため，型の詳細を見ることが出来，彼らの型スライサーは衝突するふたつの型が導入された部分を特定することが出来るという利点がある．他方で我々は構文に注目して型スライスを作成しており，型規則を与える必要がないという利点がある．

Schilling [5] は我々と同様に型制約を使用せず，既存の型推論器を用いることで型エラースライスを求めている．彼の手法としては，Haack ら [2] が型制約を用いていた部分を，プログラムの各部につけたラベルに対応させている．具体的には，プログラムの各部につけたラベルを集合として，その集合からプログラムを作成する．あるラベルが集合に存在しない場合には，そのラベルの部分は我々の手法における置き換えられたプログラム □ や @ のようなプログラムに変換する．そして作成されたプログラムをコンパイラの型推論にかけることで，それが型エラースライスか否かを判定している．ラベルを型制約のように扱うことで，最小限のラベルの組み合わせを求めることができる．Schilling [5] と我々との手法としての差異は，Schilling はプログラムを徐々に増やすことで，我々はプログラムから徐々に取り除くことでスライスを求めていることである．

これら型エラーのスライスを求める手法 [2, 5] と我々の手法の求められるスライスの比較は次のようになる．極小な型エラースライスがひとつしか存在しない場合，本手法と既存手法で得られるスライスは同じである．よって，違いは複数の極小な型エラースライスが存在する場合である．我々

の手法では，得られるスライスに局所性という制限を加えている．そのため，複数の極小な型エラースライスが存在する場合，得られるスライスは少ない可能性もあるが，ある程度有用なスライスのみが得られていると考えられる．また，既存手法で複数のスライスを見つけるためには，複数回実行する必要があったが，本手法では一回で複数のスライスを見つけることが可能である．

デバッグ　型エラーに関連する研究では，まずエラーメッセージ改良の研究が挙げられる．Wand [11] はどのように型変数が例化されたかという履歴を追跡し，型エラーが起こった場合にはどのように衝突したかという履歴を見せることを可能にした．また，Lee と Yi は型推論自体を変更し，型推論の際に周りのプログラムからどのような型が期待されるかを持ち歩くことで，Algorithm W よりも小さい範囲でエラーメッセージを出す Algorithm M [4] を提案した．Heeren と Hage [3] は制約ベースの型推論を使うことでエラーメッセージを改良した．これらの型エラーメッセージはプログラマに便利であるが，残念ながらひとつのエラーメッセージでユーザが実際に直したい場所を特定するのは不可能である．

ユーザの実際に直したい箇所を特定することを目的とした研究はいくつか挙げられる．Stuckey と Sulzmann，Wazny らは CHRsolving を使用した型推論によってエラーの原因を見つける主張 [8] を提案した．彼らはある式がなぜそのような型になったか説明することを目的とした型デバッガ Chameleon を実装した．Chitil [1] は合成的に型推論を行って最汎型木を作成することで，Haskell のサブセットを対象とした対話的な型デバッガを作成した．我々は先行研究 [9, 10] で，コンパイラの型推論を用いて最汎型木を作成することで型デバッガで型推論をする必要をなくし，OCaml のほぼフルセットを対象として型デバッガを作成した．本研究はこれら [1, 9, 10] の型エラースライスの使用による改良と位置づけられる．

8　まとめと今後の課題

本論文ではコンパイラの型推論器を使うことによって，型規則を必要としない型エラースライスの作成手法を提案した．これにより型デバッガでの質問回数減少などの応用への道が開ける．具体的には対象言語の構文に □ と S @ S という特殊な型規則をもつ構文を導入し，対象言語のプログラムを出来るだけそれらの構文で置き換えることでスライスを作成した．また，型エラースライスの性質として局所性を提案した．本節で本論文をまとめ，今後の方向性とその課題について述べる．

本論文のまとめ　先行研究 [2] は型規則を用いて型制約をベースに型エラースライスを作成していたが，本研究では構文をベースに型エラースライスを作成する方法を提案した．これによりコンパイラの型推論が使えるようになり，型規則なしで型スライサーを実装することが出来た．本研究の利点はコンパイラの型推論との差異が生じないことである．ただし，対象言語の知識がある程度必要となる．少なくとも束縛に関しては，どの構文が束縛を持つか，その束縛の対象範囲はどこまでかなどを知っている必要がある．束縛を正しく扱わないと，意味的に異なるプログラムを作成してしまうため，これは残念ながら避けられない．また，型スライサーを実装する上では型規則を知っている必要はないが，型規則を知っていると効率化が出来る場合がある．[4]

本研究の型スライサーで作成された型エラースライスはそのままデバッグに使用できる．型デバッガと型エラースライスは同じデバッグを目的とした技術であるが，これまであまり融合して使われてこなかった．型エラースライスは自動的にデバッグの対象範囲を狭められ，型デバッガはユーザ自身で型エラーを特定できる．これらを共存させることで，よりユーザに親切なデバッグを行うことが出来ると考えている．

[4] たとえば cut の段階で □@M_1 という形になったならば，cut の時点で □ @ M_1 にする効率化が考えられる．しかしこれは関数適用の型規則に関する知識を用いているため，本手法では使用していない．

今後の方向性とその課題　本節では，今後本研究をどのように発展させるかについて述べる．

まずは拡張的な方向性が挙げられる．型スライサーを拡張し，OCaml のフルセットを対象言語として実装を行うことによって，先行研究 [9, 10] のデバッガに載せることが可能になる．OCaml のフルセットに対応する場合に考えなくてはいけない点は現段階でふたつ存在する．ひとつめはパターンである．パターンは型規則を持っているため，4 節の不要中間ノードの削除に該当するが，同時に束縛でもあるため，単純に削除してよいものではない．これにはパターンを変換し，型規則を外して束縛だけ残すなどの処理が必要になると考えられる．型注釈もパターンの一種であるため，同様に扱うことが出来そうである．ふたつめは型定義やレコード定義などの定義である．これらの定義を使用する場合でも，コンテキストとして扱うことでコンパイラの型推論器が適切に型を求めてくれる．これはコンパイラの型推論器を使う利点でもある．これらの定義も一種の束縛であるため，不必要ならば束縛の削除が必要になる．その場合，定義ごとに束縛を削除する関数が必要になる．

ふたつめの方向性として，正当性の証明が挙げられる．本論文では 3 節の変換 cut に関して健全性・極小性の検証を行った．4 節の変換 $remove_node$ に関してもそれらの証明を行いたい．また両方の変換を行った後の局所性を示したいと考えている．

最後の方向性としては，他のエラースライスへの応用である．本研究ではコンパイラの型推論器を用いて，型エラースライスを作成した．この考えを応用して，他のスライスが作成出来るかを考察したい．具体的には，コンパイラの構文解析器を使用して構文エラースライスが作成出来るのか，インタプリタと (インタプリタ自体は正否を返さないため) ユーザの意図した結果を用いて意味的エラーの原因となり得るプログラムのスライスが作成出来るのかなどである．現段階では，構文エラースライスでは組み合わせが多いために計算量が問題になり，意味的エラーでは得られるスライスを元のプログラムに反映する方法が問題になりそうだと考えている．

謝辞　有益なコメントを寄せてくだ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 6th ACM SIGPLAN International Conference on Functional Programming (ICFP’01)*, pp. 193–204 (2001).

[2] Haack, C. and Wells, J. B.. “Type Error Slicing in Implicitly Typed Higher-Order Languages.” *Science of Computer Programming - Special issue on 12th European symposium on programming (ESOP’03)*, Volume 50 Issue 1-3 (2004).

[3] Heeren B. Hage, and J.. “Parametric Type Inferencing for Helium,” Technical Report UU-CS-2002-035, Utrecht University, 2002.

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

[5] Schilling, T. “Constraint Free Type Error Slicing,” *Proceedings of the 12th international conference on Trends in Functional Programming (TFP’ 11)*, pp. 1–16 (2012).

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

[7] Silva, J., and Chitil, O.. “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).

[8] Stuckey, P. J., Sulzmann, M. and Wazny, J.. “Interactive type debugging in Haskell,” *Proceedings of the 2003 ACM SIGPLAN Workshop on Haskell*, pp. 72–83 (2003).

[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”, コンピュータソフトウェア, vol.30, no.1, pp. 180–186 (2013).

[10] Tsushima, K., and Asai, K.. “An Enbedded Type Debugger,” *Proceedings of the 24th International Workshop on Implementation of Functional Languages (IFL’12)*, to appear in LNCS, Springer (2013).

[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).