実用的な型エラースライサーの提案と評価

脇川 奈穂[1], 対馬 かなえ[2]

[1] お茶の水女子大学
wakikawa.naho@is.ocha.ac.jp

[2] 国立情報学研究所
k_tsushima@nii.ac.jp

概 要

本論文では，コンパイラの型推論を用いた効率的な型エラースライス手法の提案と評価を行う．型エラースライスとは，型エラーのプログラム中から型エラーに関する箇所のみを抜き出したプログラムであり，型エラーのデバッグにおいて大いに役に立つ．本論文では既存研究を改良して，効率的に型エラースライスを生成するために，ふたつの方法を提案する．ひとつめは最も大きな子ノードを選んで抽象化する手法，ふたつめはプログラムをおおよそ半分に分けるような部分木を選んで抽象化する手法である．本論文では，初心者を対象とした授業のプログラムなどに対して実験を行い，効率に重きを置いて考察を行う．具体的には，人工的に作った型エラーのプログラムや授業での実際のプログラムを用いて，既存研究の単純な手法と提案手法を，型推論の回数や速度，元のプログラムのサイズに注目して比較する．結果として，多くの場合には本論文のアルゴリズムが高速であったが，既存研究の方が速い場合もあった．

1 はじめに

OCaml や Haskell などの関数型言語は，強い型付き言語であり，実行前に型推論を行い，型の不整合に関するエラー (型エラー) を取り除くことができる．演算子を含むすべての式が一意な型を持つ．また，OCaml では算術演算子は整数と実数で別々に，比較演算子は多相型によって 2 引数が「任意だが同一の型」を持つように定義されている．例えば，以下のようなプログラムを OCaml で実行してみよう．

```
let pow x n = x ^ n in pow (1 + 2) 3;;
```

ユーザは，pow が 2 つの整数値を受け取り，戻り値としてべき演算の結果を返す関数であると意図して定義したと仮定する．実際は，OCaml では^は文字列の結合演算子であることから，第一引数，第二引数，戻り値はいずれも string 型となり，関数呼び出しで渡した (1 + 2) と型が衝突する．型エラーはこのような型の衝突によって発生する．

コンパイラは型の衝突が発生した式に下線を引き，以下のようなエラーメッセージを出力する．コンパイラが示すエラーメッセージは，プログラマの意図にあった型エラーの原因を指摘するとは限らず，結果として自力で型エラーの原因を探さなければならないことが多々ある．実際このプログラムでは，このエラーメッセージはユーザの直したい箇所を指摘していない．ユーザが演算子^の型を誤って認識していることが原因であるため，型エラーの原因として (1 + 2) ではなく，^を示したほうが適切である．

```
Error: This expression has type int but an expression was expected of type
         string
```

1.1 型エラーデバッガ

このような問題を解決するために，対馬らはコンパイルの型推論を利用した型エラーデバッガ [5, 7] を提案し，実装した．このデバッガには，特徴として

- 型推論の際に最汎型木 [1] を用いて，アルゴリズミックデバッギング [4] を行い，型の衝突が発生している構文を特定する．
- 対話的に質問を繰り返すことによって，型エラーの原因を特定する．

といった点が挙げられる．デバッガは最汎型木が持つノードを対象に質問を行うが，対象となり得る式が必ずしも型エラーに起因するとは限らない．そこで，著者らは型エラースライス手法 [6, 9] を実装し，型エラーの原因となり得ない式の質問を削減するように型エラーデバッガの挙動を改良した [9]．

1.2 型エラースライス

型エラースライスとは，型エラーとなるプログラム中からある型エラーに関係のある箇所のみを抜き出したプログラムである．本論文では，OCaml の式の型エラースライスを求める関数を型エラースライサーと呼ぶ．型エラーに関係のない箇所は，型制約のない式 □ への置き換えによって抽象化される．極小な型エラースライスであれば，スライスに含まれる式全てが型エラーの発生に必要であり，デバッガの候補から型エラーに関係のない箇所をあらかじめ除外できる．

前述のプログラムの場合，関数の定義と呼び出しのうち第二引数に関する式は，抽象化しても型エラーが発生する箇所は変わらない．また，第一引数として渡した式が int 型であることを示すには，演算子+のみで十分である．よって，プログラムを著者らが実装した型エラースライサーに渡すと以下のような結果が得られる．

(スライス前)		(スライス後)
`let pow x n = x ^ n in pow (1 + 2) 3`	→	`let pow x n = x ^ □ in pow (□ + □) □`

1.3 型エラースライサー

既存研究 [6, 9] の型エラースライサーでは，コンパイラの型推論器を利用して型エラーになるかどうかの判定を繰り返す．アルゴリズムは再帰的な木構造の走査であるとみなせる．型エラースライサーは単純型付きラムダ計算を追加したような言語を対象とし，本論文では図 1 に示したようにリスト構造を明示した．

$(S : slice)$	:=	c	(定数)	$(\tau : type)$	:=	$int, bool, \ldots$	(型定数)
	|	x	(変数)		|	α	(型変数)
	|	$\lambda x.S$	(関数抽象)		|	$\tau * \ldots * \tau$	(タプル型)
	|	$@S \ldots S$	(関数適用)		|	$\tau \to \tau$	(関数型)
	|	$let\ x = S\ in\ S$	(局所関数定義)		|	$\tau\ list$	(リスト型)
	|	$(S, \ldots, S)$	(タプル)				
	|	$[\,]$	(空リスト)				
	|	$S :: S$	(非空リスト)				
	|	□	(ホール)				

図 1: スライスの主な構文と型

既存研究の手法の詳細は参考文献の論文 [6] を参照されたい．この手法はスライシング対象となる部分式を，基本的に深さ優先探索に即した順序で選び，ノードをひとつずつ抽象化して型エラー

になるかどうかを判定している．はじめに，スライシング対象の式に対して子ノードをひとつ抽象化したような式の集合を作り，その中から型エラーになる式を探す．該当する式が見つかった場合，抽象化された子ノードは不要であり，型エラーになる式を新たな対象として集合を作り直す．この処理は該当する式が見つからなくなるまで繰り返す．次に，抽象化されずに残った子ノードを頂点とする部分式を対象として，これらのスライスを上記の処理を適用して順に求める．葉ノードは型エラーに不要であれば抽象化された式を，必要であればそのまま確定する．内部ノードは全ての子ノードが確定した時点で親ノードも同様に確定し，全てのノードの確定をもって式全体の型エラースライスが求められる．

実行例として，前述の式の型エラースライスを求める過程を以下に示す．各ステップには，その時点での全体の構造（現在の状態）と型エラースライスを求めたい部分式（スライシング対象）を明記した後，スライシング対象の式から作られた集合の要素を抽象化される子ノード（子ノード）と，型推論によって抽象化しても型エラーになると判断された要素があればひとつだけ記載する（抽象化されるもの）．

step 1 :	（現在の状態）	`let pow x n = x ^ n in pow (1 + 2) 3`
	（スライシング対象）	現在の状態と同じ
	（子ノード/抽象化されるもの）	`(fun x n -> x ^ n), (pow (1 + 2) 3)` / なし
step 2 :	（スライシング対象）	`pow (1 + 2) 3`
	（子ノード/抽象化されるもの）	`pow, (1 + 2), 3` / `3`
step 3 :	（現在の状態）	`let pow x n = x ^ n in pow (1 + 2) □`
	（スライシング対象）	`pow (1 + 2) □`
	（子ノード/抽象化されるもの）	`pow, (1 + 2)` / なし
step 4 :	（スライシング対象）	`(1 + 2)`
	（子ノード/抽象化されるもの）	`(+), 1, 2` / `2`
step 5 :	（現在の状態）	`let pow x n = x ^ n in pow (1 + □) □`
	（スライシング対象）	`(1 + □)`
	（子ノード/抽象化されるもの）	`(+), 1` / `1`
step 6 :	（現在の状態）	`let pow x n = x ^ n in pow (□ + □) □`
	（スライシング対象）	`(□ + □)`
	（子ノード/抽象化されるもの）	`(+)` / なし
step 7 :	（スライシング対象）	`fun x -> n -> (x ^ n)`
	（子ノード/抽象化されるもの）	`fun n -> (x ^ n)` / なし
step 8 :	（スライシング対象）	`fun n -> (x ^ n)`
	（子ノード/抽象化されるもの）	`(x ^ n)` / なし
step 9 :	（スライシング対象）	`x ^ n`
	（子ノード/抽象化されるもの）	`(^), x, n` / `n`
*step*10 :	（現在の状態）	`let pow x n = x ^ □ in pow (1 + 2) □`
	（スライシング対象）	`x ^ □`
	（子ノード/抽象化されるもの）	`(^), x` / なし

また，OCaml には structure と呼ばれる構造によって，プログラム中に複数の関数を定義し，以降で呼び出せるようになっている．一般的に式は関数として定義される場合が多いが，全ての関数が型エラーに起因するとは限らない．そのため，実際は型エラースライサーを実行する前に，structure の 1 要素に含まれる式そのものを抽象化した際の型推論を行っている．

1.3.1 特徴と問題点

既存研究の型エラースライス手法の特徴は実装の容易さにある．しかしながら，プログラムのサイズや構文によっては型推論の回数が多くなり，結果が得られるまでに時間がかかる．問題の原因として

- ひとつのノードを抽象化するたびに型推論を行う．
- 特定のノードが何度も抽象化の対象となり得る．

といった点が挙げられる．前者は `(1 + 2)` に残る部分式が 1 つだけ残るため，*step* 4 と *step* 5 で選んだ子ノードを同時に抽象化すると型推論が一度で済む．型エラースライシングの実行時間のうち型推論が占める割合は高く，ノードの数が多いほど型推論の回数が増加する．この傾向は型エラーの原因となり得る部分式が深い位置にある場合に，さらにリスト構造のような著しいノードの偏りが見られる場合に強く現れる．後者は *step* 2 と *step* 3 で `pow, (1 + 2)` のように複数回抽象化される候補が発生する場合に起こり得る．型エラーになる要素が集合の末尾にある場合，他の要素全ての型推論が行われるため無駄があるといえる．

1.3.2 解決方法

式は多数のノードによって構成され，ノード数が増加するほどサイズも大きくなり，型の衝突が発生する確率も上昇する．一方で，ノード数に対する型エラースライスとして残ったノード数の割合はあまり高くならない．型エラーの直接の原因となり得るノードは数個程度で，定数や変数といった葉ノードを含む場合がほとんどである．このことから，スライシング対象の式の子ノードは，ノード数の合計が多いほど抽象化されずに残る数の割合が低く，各々の型エラーに起因する確率はサイズに依存すると仮定した．仮定のもとに，筆者らは

- 必要に応じて複数の式を同時に抽象化する
- サイズを基準に抽象化する式を選ぶ

といった変更を加えた実用的な型エラースライサー [8] を提案し，実装した．本論文ではアルゴリズムを再定義するとともに多くのプログラムを対象に実験を行い，効率を重視して各スライサーを評価する．また，本節以降はノード数をサイズと呼ぶ．

2 提案する型エラースライス手法

本節では，プログラムのサイズを考慮した効率的な型エラースライス手法について述べる．また，提案手法と比較し，評価を行うために既存研究の手法のスライサーを *slicer*0 と呼ぶ．

対象とする構文を図 2 に示した．サイズは部分式中に含まれるホールを除いたノードの総数と同義で，式の状態にあわせて変化する．また，複数のノードを抽象化するにあたって，抽象化の必要性の判定に位置情報を用いる．例えば，以下に示した下線部の位置情報は `line 1, characters 27-34` である．

```
let pow x n = x ^ n in pow (1 + 2) 3
```

前述の位置情報は,「ファイル中の 1 行目の,（行頭を 0 文字目とする）27 文字目から始まり 34 文字目で終わる」ことを表している．OCaml の式がネスト構造であることから，位置情報の包含関係の有無を抽象化の指標として利用する．また，各ノードはスライシングの進行によって位置情報以外は常に変化し得るため，スライスの等価判定は位置情報とノードの状態が一致した場合にのみ真であると定義する．

$$
\begin{array}{lcll}
(S : slice) & := & c^n & (n : size) := n \quad \text{(プログラムのサイズ)} \\
& | & x^n & (\tau : type) := \quad \text{(サイズを考慮しない場合と同じ)} \\
& | & \lambda x^n.S & \\
& | & @^n S \ldots S & \\
& | & let^n\ x = S\ in\ S & \\
& | & (S, \ldots, S)^n & \\
& | & [\,]^n & \\
& | & S ::^n S & \\
& | & \Box &
\end{array}
$$

図 2: サイズを考慮したスライスの構文

プログラムのサイズを考慮するにあたって，複数のノードに対して特定の処理を適用したり，サイズの計算に子ノードのサイズを求める必要があることから，map 関数 map_slice と fold 関数 $fold\_slice$ を定義した．map_slice は，$slice \to slice$ 型の関数 f とノード S を受け取り，S の子ノードに対して個々に f を適用する．$fold\_slice$ は，$\alpha \to slice \to \alpha$ 型の関数 f，α 型の初期値 $init$，ノード S を引数として受け取り，S の子ノード $s_1, s_2, \ldots, s_n$ に対して $fold\_slice[\![(f, (\ldots fold\_slice[\![(f, init, s_1)]\!] \ldots), s_n)]\!]$ のように関数適用の結果を畳み込む．

map_slice と $fold\_slice$ の実行例を以下に示す．$abstLeaf$ は引数が葉ノードであれば抽象化し，そうでなければそのままの状態で式を返す 1 引数関数，$isLeaf$ は引数に真偽値とノードを渡し，第一引数が $true$ かつ第二引数が葉ノードであるかどうかを返す 2 引数関数とする．

$$
\begin{array}{rcl}
map\_slice[\![(abstLeaf, (1, (2, 3), 4))]\!] & = & (\Box, (2, 3), \Box) \\
fold\_slice[\![(isLeaf, true, (1, (2, 3), 4))]\!] & = & false
\end{array}
$$

探索の優先順位以外に大きな差異はないので，図 3 に示したような型推論に必要となる関数を定義したライブラリを用意し，ライブラリ中の関数を利用して型エラースライサーを実装した．

$$
\begin{array}{lcl}
error? & : & slice \to bool \\
error?[\![S]\!] & = & \text{型推論を行い，スライスが型エラーになるか否かを返す} \\
set\_slice & : & slice * bool \to slice \\
set\_slice[\![(S^n, sel)]\!] & = & if\ (sel)\ then\ \Box\ else\ S^0 \\
consist? & : & slice * slice \to bool \\
consist?[\![(S_1, S_2)]\!] & = & S_1 \text{と} S_2 \text{の位置情報が一致しているか否かを返す} \\
make\_cxt2 & : & slice \to ((slice \to slice) * bool * slice \to slice) \\
make\_cxt2[\![S]\!] & = & fun\ (cxt, sel, S') \to \\
 & & let\ cxt' = fun\ S_n \to let\ sel' = consist?[\![(S', S_n)]\!]\ in \\
 & & if\ (S' <> S_n\ \&\&\ sel')\ then\ S' \\
 & & else\ if\ ((sel\ \&\&\ sel')||(not\ sel'))\ \ then\ S_n \\
 & & else\ set\_slice[\![(S_n, true)]\!]\ in\ cxt[\![map\_slice[\![(cxt', S)]\!]]\!] \\
make\_cxt1 & : & ((slice \to slice) * bool * slice \to slice) * \\
 & & (slice \to slice) * bool \to (slice \to slice) \\
make\_cxt1[\![(cxt2, cxt1, sel)]\!] & = & fun\ S' \to cxt2[\![(cxt1, sel, S')]\!]
\end{array}
$$

図 3: サイズを考慮したスライスのライブラリ

$error?$ はノードを受け取り，型エラーが発生するか否かを判定する．以降，型エラーが発生した

場合を illtyped，型がついた場合を welltyped と呼ぶ．与えられたノードが illtyped であれば真を，welltyped であれば偽を返す．抽象化とノードの確定は同一の関数 set_slice で行う．この関数はノードと真偽値を受け取り，真であった場合にはノードを抽象化し，偽であればノードを型エラースライスとして確定する．抽象化の判定には $consist?$ を用いる．$consist?$ はノードを 2 つ受け取り，これらの位置情報が一致していれば $true$ を，一致していなければ $false$ を返す．

ここまでで述べた関数を組み合わせて，全体の式が型エラーになる状態を保ちながら，抽象化を繰り返して極小な型エラースライスを求める．現在の状態を記憶する方法として，コンテキストを利用する．コンテキストとは，あるノードを除く他のノードの情報を意味する．提案手法では，全体の式が得られるコンテキストと部分式が得られるコンテキストを組み合わせて型エラーの判定を行う．全体の式のコンテキストでは，指定された真偽値に応じた部分的な抽象化を行い，その結果を部分式のコンテキストで畳み込む．部分式のコンテキストは，既存研究のコンテキストと同様に与えられたノードをコンテキストに埋め込む．

$make\_cxt2$ はノード S を受け取り，戻り値として全体の式を返すコンテキストを返す．全体の式のコンテキストは，任意の部分式のコンテキスト，抽象化するノードの数が複数か否かを指定する真偽値，抽象化の基準とするノード S' を受け取り，単数が指定された場合には S' のみを，複数が指定された場合には S' とその先祖以外を抽象化する．S' と位置情報のみ一致するノードを発見した場合には，該当するノードを S' に置き換える．$make\_cxt1$ は $make\_cxt2$ によって生成された関数 $cxt2$ と，ノードを除く $cxt2$ に与える引数を受け取り，新たな部分式のコンテキストを生成して返す．なお，後述する 2 手法はノード S を基準として，以下のルールに従って新たなスライシング対象のノードとコンテキストを設定する．

1. $cxt2[\![(cxt1, true, S)]\!]$（複数の抽象化：$S$ 以外に抽象化され得るノードが存在する場合のみ）

   illtyped: S が対象．コンテキストを $make\_cxt1[\![(cxt2, cxt1, true)]\!]$ に変更．

   welltyped: 次に進む．

2. $cxt2[\![(cxt1, false, S)]\!]$（単数の抽象化）

   illtyped: $cxt2[\![(fun\ x \to x, false, S)]\!]$ が対象．$cxt1$ はそのまま．

   welltyped: S が対象．コンテキストを $make\_cxt1[\![(cxt2, cxt1, false)]\!]$ に変更．

式 `1 - 2 = false` の型エラースライスを求める過程において，現在のスライシング対象が `1 - 2` かつ `false` が既に型エラースライスに必要であることが確定済で，基準となるノードとして `1` が選択されていると仮定する．このような場合には以下に示したようなコンテキストが定義されている．

$$
\begin{array}{lcl}
cxt1[\![S^n]\!] & = & (@ =^1\ \ S^n\ \ false^0)^{n+2} \\
cxt2[\![(cxt1, true, 1^1)]\!] & = & (@\square\ (@\square\ 1^1\ \square)^2\ \ false^0)^3 \\
cxt2[\![(cxt1, false, 1^1)]\!] & = & (@ =^1\ \ (@ -^1\ \ \square\ 2^1)^3\ \ false^0)^5 \\
cxt2[\![(cxt1, false, 1^0)]\!] & = & (@ =^1\ \ (@ -^1\ \ 1^0\ 2^1)^3\ \ false^0)^5 (*\text{該当ノードを置き換える}*)
\end{array}
$$

2.1 最も大きい子ノードを選ぶ方法

最も大きい子ノードを選ぶ方法では，抽象化される式は単純なアルゴリズムと同様に，子ノードのみに限られるが，複数の子ノードを同時に抽象化する場合を考えなければならない．この点を考慮して定義したアルゴリズムを図 4 に示した．

$children$ はノードを受け取り，スライシングの基準となり得る子ノードをサイズに対して降順のリストとして返す．$search1$ は子ノードのリスト，部分式のコンテキスト $cxt1$，全体の式のコンテキスト $cxt2$ を受け取り，先頭の要素 S を基準に型エラーの判定を行い，基準となるノードとコンテキストを変更する．末尾 $slices$ が空であれば，複数の抽象化を指定する場合の判定は行わない．リストが空であれば，$cxt2$ で単数の抽象化を選び，位置情報のない $\square$ を与えて $cxt1$ を畳み込むことに

$$
\begin{array}{lcl}
children & : & slice \to slice\ list \\
children[\![S]\!] & = & \text{スライスのサイズが 1 以上の子ノードのリストを} \\
 & & \text{サイズに対して降順にソートして返す} \\
search1 & : & (slice\ list * (slice \to slice)* \\
 & & ((slice \to slice) * bool * slice \to slice)) \to slice \\
search1[\![([], cxt1, cxt2)]\!] & = & let\ S' = cxt2[\![(cxt1, false, \Box)]\!]\ in \\
 & & if\ (get\_size[\![(0, S')]\!] = 0)\ then\ S' \\
 & & else\ slicer1[\![(S', fun\ S'' \to S'')]\!] \\
search1[\![(S :: slices, cxt1, cxt2)]\!] & = & if\ (slices <> [] \ \&\&\ error?[\![cxt2[\![(cxt1, true, S)]\!]]\!])\ then \\
 & & \quad slicer1[\![(S, make\_cxt1[\![(cxt2, cxt1, true)]\!])]\!] \\
 & & else\ if\ (error?[\![cxt2[\![(cxt1, false, S)]\!]]\!])\ then \\
 & & \quad slicer1[\![(cxt2[\![(fun\ x \to x, false, S)]\!], cxt1)]\!] \\
 & & else\ if\ (get\_size[\![(0, S)]\!] = 1)\ then \\
 & & \quad let\ cxt1' = fun\ S' \to set\_slice[\![(S', false)]\!]\ in \\
 & & \quad slicer1[\![(cxt2[\![(cxt1', false, S)]\!], cxt1)]\!] \\
 & & else \\
 & & \quad slicer1[\![(S, make\_cxt1[\![(cxt2, cxt1, false)]\!])]\!] \\
slicer1 & : & (slice * (slice \to slice)) \to slice \\
slicer1[\![(S, cxt)]\!] & = & search1[\![(children[\![S]\!], cxt, make\_cxt2[\![S]\!])]\!]
\end{array}
$$

図 4: 最も大きな子ノードを選んで抽象化する手法

よって，全体の型エラースライスが得られる．S のサイズが 0 であればアルゴリズムを終了し，それ以外の場合には再び頂点をスライシング対象に選んで再帰する．メイン関数は $slicer1$ で，ノード S とコンテキスト cxt を受け取り，S が葉ノードに相当する構文であれば抽象化せずに確定し，その他の場合は型エラーの判定に進む．このアルゴリズムで型エラースライスを求める過程の一部を以下に示す．各ステップではスライシング対象 S と基準となり得る子ノードのリスト $children[\![S]\!]$，型推論を行う式と結果を，関数ごとに記載した．また，$step\ 2(search1)$ のようにリストの要素が複数ある場合には，1 回目の型推論では複数の部分式を抽象化している．

$step\ 1$: $(slicer1)$ $S =$ `let pow x n = x ^ n in pow (1 + 2) 3` (∗初期状態∗)
$children[\![S]\!] =$ `(pow (1 + 2) 3)`^{7} :: `(fun x n -> ...)`^{6} :: []
$(search1)$ `let pow x n = x ^ n in` □ / welltyped
`let pow =` □ `in pow (1 + 2) 3` / welltyped

$step\ 2$: $(slicer1)$ $S =$ `pow (1 + 2) 3`, $children[\![S]\!] =$ `(1 + 2)`^{4} :: `pow`^{1} :: `3`^{1} :: []
$(search1)$ `let pow x n = x ^ n in` □ `(1 + 2)` □ / welltyped
`let pow x n = x ^ n in pow` □ `3` / illtyped

$step\ 3$: $(slicer1)$ $S =$ `pow` □ `3`, $children[\![S]\!] =$ `pow`^{1} :: `3`^{1} :: []
$(search1)$ `let pow x n = x ^ n in` □ □ `3` / welltyped
`let pow x n = x ^ n in pow` □ □ / welltyped (∗`pow` を確定∗)

$step\ 4$: $(slicer1)$ $S =$ `pow` □ `3`, $children[\![S]\!] =$ `3`^{1} :: [] (∗$slices = []$∗)
$(search1)$ `let pow x n = x ^ n in pow` □ □ / welltyped (∗`3` を確定∗)

$step\ 5$: $(slicer1)$ $S =$ `pow` □ `3`, $children[\![S]\!] = []$
$(search1)$ $S' =$ `let pow x n = x ^ n in` □ / 再帰の必要あり

$step\ 6$: $(slicer1)$ $S =$ `let pow x n = x ^ n in pow` □ `3` (∗バックトラック∗)
$children[\![S]\!] =$ `(fun x -> fun n -> x ^ n)`^{6} :: [] (∗$slices = []$∗)

2.2 全体の半分程度の大きさのノードを選ぶ方法

全体の半分程度の大きさのノードを選ぶ方法では，スライシングの基準となるノードの探索が部分式全体に及ぶため，コンテキストでのサイズの更新が必要となる．そこで，基準となるノードの選択とコンテキストの生成を同時に行う関数を図 5 のように定義した．

$$
\begin{array}{lcl}
choose & : & slice * (slice * slice \to slice) * \\
 & & (slice * bool \to slice) \to (slice * (slice \to bool) * \\
 & & ((slice \to slice) * bool * slice \to slice)) \\
choose[\![(S, pick, abst)]\!] & = & let\ child\ = fold\_slice[\![(pick, S, S)]\!]\ in \\
 & & if\ (child = S)\ then \\
 & & \quad (S, (fun\ S' \to S' = abst[\![(true, S)]\!]), \\
 & & \quad fun\ (cxt, sel, slice) \to cxt[\![abst[\![(sel, slice)]\!]]\!]) \\
 & & else \\
 & & \quad let\ abst'\ = fun\ (sel, slice) \to let\ m\ = ref\ 1\ in \\
 & & \quad let\ S'\ = map\_slice[\![(*\ sel \text{と位置情報に応じて子ノードを} \\
 & & \quad \text{抽象化し、抽象化されなかったノードのサイズを } m \text{ に加算する。} \\
 & & \quad *)]\!]\ in\ abst[\![(sel, S'^{(if\ (!m=1)\ then\ 1\ else\ !m\ )})]\!]\ in \\
 & & \quad choose[\![(child, pick, abst')]\!]
\end{array}
$$

図 5: 全体の半分程度の大きさのノードを選ぶ方法の補助関数

choose は，対象のノード S，全体の半分程度の大きさのノードを選ぶ関数 *pick*，現時点の全体の式のコンテキストの部分関数 *abst* を受け取り，基準となるノード，基準となるノード以外の未確定のノードの有無を判定する関数，全体の式のコンテキストを返す．*pick* は 2 通り考える．一方は，打ち切り条件を設けてサイズが最大の子ノードを返す．もう一方は，2 つのノードを比較し，より全体のサイズの半分に近いノードを返す．例えば，`(let pow x n = x ^ n in pow (1 + 2) 3)`[14] の基準となるノードには，それぞれ

*slicer*2 の場合： `(1 + 2)`[4]（3 回目のchooseで確定）
*slicer*3 の場合： `(pow (1 + 2) 3)`[7]（2 回目のchooseで確定）

が選ばれる．前者を *slicer*2，後者を *slicer*3 として，これらのアルゴリズムを図 6 のように定義した．

*search*2 は，2 つのコンテキスト *cxt*1 と *cxt*2 と，子ノードの代わりに基準となるノード S と全体の式のコンテキストに渡す真偽値 *sel* を受け取っている．$sel = true$ かつ welltyped になった場合には，$sel = false$ に変更して再帰している．戻り値は *search*1 と同様である．スライシングの終了判定は *search*1 と異なり，後述するメイン関数の中で行う．

メイン関数の *slicer*2 は，入出力の型が *slicer*1 と一致している．ノード S が 2 未満の場合は，*slicer*2′ に引数をそのまま渡し，それ以外は *choose* によって必要な関数やノードを得て，それらを *search* に渡す．複数の抽象化が必要か否かは，f に S を適用した結果によって判断し，この結果を直接 *sel* として渡す．*slicer*2′ は，コンテキストにノードを渡して，その式のサイズが 1 以上であればスライシングを続けて，0 であればスライシングを終了する．*search*3 と *slicer*3′ は，*slicer*2 が *slicer*3 に置き換えられている以外は全く同じ関数として定義した．*slicer*3 と *slicer*2 の差異は，実質的には *pick* 関数による基準となるノードの選択条件のみである．*slicer*1 と同様に，*slicer*2 と *slicer*3 それぞれの型エラースライスを求める過程の一部を以下に示す．

step 1 :
(*slicer*2) $S' =$ `(1 + 2)`[4]
(*search*2) `(let pow = □ in □ (1 + 2) □)`[6] / welltyped
(*slicer*3) $S' =$ `(pow (1 + 2) 3)`[7]
(*search*3) `(let pow = □ in pow (1 + 2) 3)`[8] / welltyped

$search2$: $(slice * (slice \to slice) * bool *$
$((slice \to slice) * bool * slice \to slice) \to slice$

$search2[\![(S, cxt1, true, cxt2)]\!] = if\ (error?[\![cxt2[\![(cxt1, true, S)]\!]]\!])\ then$
$slicer2[\![(S, make\_cxt1[\![(cxt2, cxt1, true)]\!])]\!]$
$else$
$search2[\![(S, cxt1, false, cxt2)]\!]$

$search2[\![(S, cxt1, false, cxt2)]\!] = if\ (error?[\![cxt2[\![(cxt1, false, S)]\!]]\!])\ then$
$slicer2[\![cxt2[\![(fun\ x \to x), true, S)]\!], cxt1)]\!]$
$else\ if\ (1 < get\_size[\![cxt[\![S]\!]]\!])\ then$
$slicer2[\![(S, make\_cxt1[\![(cxt2, cxt1, false)]\!])]\!]$
$else$
$let\ slice\ = cxt2[\![(cxt1, false, set\_slice[\![(S, false)]\!])]\!]\ in$
$slicer2[\![(slice, fun\ x \to x)]\!]$

$slicer2$: $(slice * (slice \to slice)) \to slice$

$slicer2[\![(S, cxt)]\!] = if\ (get\_size[\![(0, S)]\!] < 2)\ then slicer2'[\![(S, cxt)]\!]$
$else$
$let\ pick\ = fun\ (s', s) \to$
$(*fold\_slice$ によってサイズが最大の子ノードを選ぶ。
ただし、s'が初期値かつサイズが全体の半分以下はs'を、
s'が初期値かつsのサイズが1以上の場合はsを返す。$*)\ in$
$let\ (S', f, cxt2)\ = choose[\![(S, pick, fun\ (\_, x) \to x]\!]\ in$
$search2[\![(S', cxt, f[\![S]\!], cxt2)]\!]$

$slicer2'$: $(slice * (slice \to slice)) \to slice$

$slicer2'[\![(S, cxt)]\!] = if\ (0 < get\_size[\![cxt[\![S]\!]]\!])\ then\ slicer2[\![(cxt[\![S]\!], fun\ x \to x)]\!]$
$else\ cxt[\![S]\!]$

$search3$: $search2$ とほぼ同じなので割愛

$slicer3$: $(slice * (slice \to slice)) \to slice$

$slicer3[\![(S, cxt)]\!] = if\ (1 < get\_size[\![(0, S)]\!])\ then\ slicer3'[\![(S, cxt)]\!]$
$else$
$let\ pick\ = fun\ (s', s) \to (*\ fold\_slice$ によって
サイズがSの半分により近いノードを返す。$*)\ in$
$let\ (S', f, cxt2)\ = choose[\![(S, pick, fun\ (\_, x) \to x)]\!]\ in$
$search3[\![(S', cxt, f[\![S]\!], cxt2)]\!]$

$slicer3'$: $slicer2'$とほぼ同じなので割愛

図 6: 全体の半分程度の大きさのノードを選ぶ方法

	($search2$)	`(let pow x n= x ^ n in pow □ 3)`[10] / illtyped	($search3$)	`(let pow x n = x ^ n in □)`[8] / welltyped
step 2 :	($slicer2$)	$S =$ `(let pow ... in pow □ 3)`[10]	($slicer3$)	$S =$ `(pow (1 + 2) 3)`[7]
	($search2$)	省略	($search3$)	省略
step 3 :	($slicer2$)	$S =$ `(x ^ n)`[4]	($slicer3$)	$S =$ `(pow □ 3)`[3]
	($search2$)	省略	($search3$)	省略
step 4 :	($slicer2$)	$S =$ `(let pow ... in pow □ 3)`[9]	($slicer3$)	$S =$ ($*$ $slicer2$ と同じ $*$)

3　実験と評価

提案手法のアルゴリズムの実用性を検証するために，各手法について様々なプログラムを対象に型エラースライシング中の型推論に要した時間の総和（実行時間）と型推論の回数を測定し，結果を比較およびに評価した．1 回の型推論に要した時間は，*error*? の呼び出しの前後での CPU 時間の差分を意味する．実際のプログラムは structure が複数の要素を持つ場合が多々あるが，本論文では要素ごとの式の測定値の総和を，ひとつのプログラムにおける実験結果とみなす．さらに，要素中の式そのものが全く型エラースライスに関係ない場合もあり得るため，型エラースライスを求める前に要素中の式が必要かどうかの判定を行った．

実行時間の測定やデータの書き込みと出力は，以下の実験環境にて型エラースライサー中で行い，スライス前の式のサイズの総和，実行時間，型推論の回数をプログラム毎に記録した．さらに，同一項目での既存研究と提案手法の比を「効率」と定義し，数値の大小によって実用性の高低を判断した．

OS:　macOS Sierra バージョン 10.12.6

プロセッサ:　2.8GHz Intel Core i5

メモリ:　8 GB 1600 MHz DDR3

実験の対象となるプログラムを下記の通りに分類し，表 1 に示したようなデータを統計的に分析して評価した．お茶の水女子大学では，OCaml の初学者を対象とした「関数型言語」という授業を開講している．授業で使用する計算機室の PC の OCaml には型エラーデバッガが組み込まれており，デバッガが起動した際のエラーログとプログラムを採取している．

1. ノードに偏りがあるプログラム（算術演算，比較演算，リスト構造）
2. 「関数型言語」のプログラム
3. サイズが大きいプログラム

プログラムの種類	ノードに偏りがあるプログラム			「関数型言語」のプログラム	サイズが大きいプログラム		
	算術演算	比較演算	リスト構造		comp	model	infer
総数	1274	2500	1274	4524	10	10	10
サイズの平均	99.08	35.62	154.5	129.9	2257	527.1	849.8
サイズの分散	35.62	61.2	151	241.8	2.221	290.0	1.989

表 1: 実験データに関する情報

エラーログを分析した結果，デバッガが示す型エラーの原因となる構文の多くが関数適用であり，多くのプログラムのテストケースとして比較演算子=を含んでいた．既存研究の手法の問題点から，ノードに偏りがあるプログラムほど提案手法の効率が向上すると仮定した．そこで，関数適用およびにリスト構造のプログラムをノードに偏りがあるプログラムとして人工的に生成し，これらを対象とした実験では構文による差とバランスの偏りに着目して評価した．また,「関数型言語」で作成するプログラムは，ごくごく簡単なサイズが小さいものから複雑でサイズが大きいものまで幅広く存在するため，これらを対象に相対的に実行時間を比較し評価した．さらに，OCaml の初学者が自力で作成するプログラムのサイズは大きくなりにくいため，別途サイズが大きいプログラムを対象に実験し,「関数型言語」のプログラムと同様に評価した．分析結果は表として掲載するが，プログラムの分類ごとに検証すべき点が異なるため，分析結果の項目についての説明をあわせて行う．なお，表中の数値は基本的に 4 桁の有効数字で四捨五入している．

3.1　ノードに偏りがあるプログラム

実験によって得られたデータから，型推論の実行時間と，実行回数およびにプログラムのサイズとの相関係数を求めた．スライサーの性能は，1 回あたりの型推論に要する時間がプログラム毎に一定であると仮定し，実験対象となるファイルの規則性に応じた時間計算量によって推測した．加えて，実行時間との相関は散布図を作成する．既存研究の手法では，n 個の子ノードを直下に持つノードの型推論の回数が，m 個のノードが抽象化されずに残る場合に最大 $(m+1)*(n-m+1)+1$ となり，抽象化されずに残るノード数の最大は n である．よって，計算量が $O(n^2)$ 程度になると考えられる．提案手法では，プログラムを分割するという点から，計算量が最悪でも $O(n\ log\ n)$ 程度になると考えられる．そのため，累乗近似による近似曲線 $v = a * 10^{-4} * x^n$ と決定係数 R^2 を求めて，近似曲線を散布図に重ね合わせて示した．

3.1.1　算術演算

一般的な演算子である + は $int \to int \to int$ 型として定義されていて，構文解析によってノードが偏り，1 つのノードのみが極端に大きくなりやすい．実験の対象となるプログラムは

```
(true + 1), (1 + true), (true + 1 + 1), (1 + true + 1), (1 + 1 + true), ...
```

のように，演算子の項のひとつの型を int ではなく $bool$ とした式を，項の数と型を $bool$ とする項の位置を様々な組み合わせで生成した．これらのプログラムは第一引数に相当するノードのサイズが大きく，木の深さが 1 の個数と同じになる．実験データの分析結果を表 2 に，プログラムのサイズと型推論の実行回数の散布図を図 7 に，プログラムのサイズと実行時間の散布図を図 8 に示した．なお，型推論の回数の最小値は，$slicer0$ が 3，$slicer1$ が 4，$slicer2$ と $slicer3$ が 6 であった．

(効率)	0	1	2	3
時間	(1.000)	1.832	4.448	4.472
回数	(1.000)	1.717	4.853	4.809
(相関係数)	0	1	2	3
回数	0.9638	0.9635	0.4595	0.6312
サイズ	0.6386	0.6356	0.6606	0.6747
(近似曲線)	0	1	2	3
a	0.2096	0.2230	1.000	1.000
n	1.757	1.612	1.055	1.100
R^2	0.4979	0.4953	0.6309	0.6380

表 2: 実験データの分析結果 (算術)

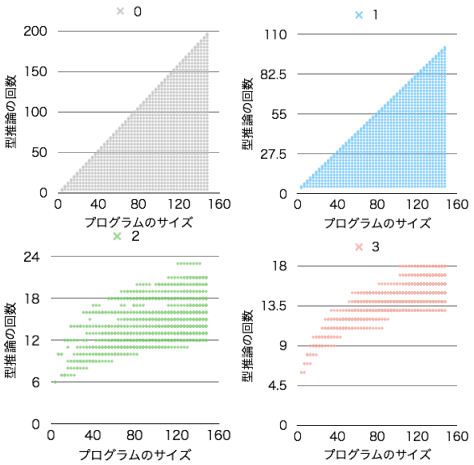

図 7: サイズと回数の散布図 (算術)

算術演算を中心としたプログラムでは，提案手法はいずれも既存研究の手法に比べて効率が高く，特に $slicer2$ や $slicer3$ のように全体の半分程度の大きさのノードを選ぶ方法が有用であった．$slicer1$ はサイズ毎の型推論の回数が $slicer0$ に比べて最大値が半分程度になったが，最小値と最大値の差が線形に増加する点は変わらない．それに比べて，$slicer2$ と $slicer3$ は型推論の回数は対数的に増加し，最小値はやや高いが最大値が大幅に抑えられたため，安定して高速であった．計算量は近似式より，$slicer0$ と $slicer1$ が $O(n^2)$，$slicer2$ と $slicer3$ が $O(n)$ と考えられる．散布図を見ると，サイズ毎の実行時間の最低値には大きな差はないが，最高値が異なっていることがわかる．$slicer2$ と $slicer3$ は，決定係数 R^2 からこれらのアルゴリズムに大きな差はないが，$slicer0$ と $slicer1$ よりプログラムのサイズ毎の実行時間の幅が小さく，優れているといえる．型推論の回数と実行時間の相関は，$slicer0$ と $slicer1$ では非常に強いが，抽象化されるノードが子ノードに制限されない $slicer2$

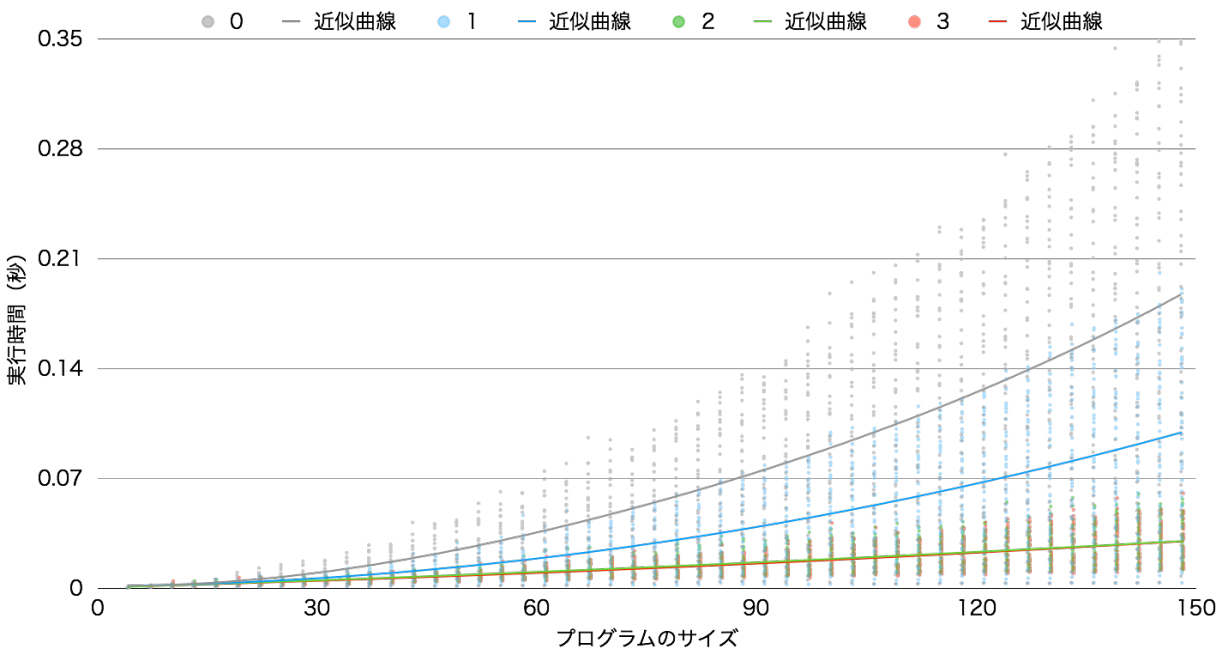

図 8: サイズと時間の散布図 (算術)

と $slicer3$ ではあまり強くない．プログラムのサイズと実行時間の相関はいずれもやや強いが，アルゴリズムによる大きな差は見られない．よって，全体のサイズより型エラーに起因する位置のほうがより重要であるといえる．具体的には，$slicer0$ と $slicer1$ は，抽象化するノードの候補が直下の子ノードに限られるため，型エラーを起こすノードが深い位置にある場合に型推論の回数が多くなり，効率に大きな差が現れると考えられる．

3.1.2 比較演算

主に条件分岐で用いられる比較演算子 = は $\alpha \to \alpha \to bool$ 型として定義されていて，1 つのノードのみが極端に小さくなりやすく，型エラーに起因するノードが浅い位置に偏りやすい．実験の対象となるプログラムは

```
(1 = 1.0), (1 + 1 = 1.0), (1 = 1.0 +. 1.0), (1 + 1 = 1.0 +. 1.0), ...
```

のように，2 項にそれぞれ型が異なる式を与えて生成した．これらのプログラムでは，比較演算が持つ部分式のうち関数部分が演算子=，2 つの引数は算術演算とほぼ同様の構造を持つ．実験データの分析結果を表 3 に，プログラムのサイズと型推論の実行回数の散布図を図 9 に，プログラムのサイズと実行時間の散布図を図 10 に示した．なお，型推論の回数の最小値は，$slicer0$ が 3，$slicer1$ が 5，$slicer2$ と $slicer3$ が 6 であった．

比較演算を中心としたプログラムでは，提案手法のほうが実行時間の効率はやや高いが，型推論の回数は増加した．よって，提案手法のほうが 1 回の型推論に要する時間が短くなったと考えられる．各手法の差は小さいが，複数の式を抽象化することによる効果は得られた．計算量は近似式より，$slicer0$ と $slicer1$ が $O(n)$，$slicer2$ と $slicer3$ は平均的には $O(n \log n)$ と考えられるが，サイズ毎の最大値が線形に増加していることから $O(n)$ が妥当と言える．散布図を見ると，$slicer0$ と $slicer1$ はデータのばらつきが小さい．多くの場合には $slicer0$ と $slicer1$ より高速だったが，少数だが極端に実行時間が遅くなったプログラムがあったため，一概に効率的なアルゴリズムであるとはいえない．$slicer0$ と $slicer1$ はプログラムのサイズと実行時間に非常に強い相関があった．比較演算子が多相関数であるため，型推論の回数と実行時間の相関が弱いと考えられる．逆に，$slicer2$ と $slicer3$ は

(効率)	0	1	2	3
時間	(1.000)	1.240	1.342	1.318
回数	(1.000)	0.6964	0.3858	0.3931
(相関係数)	0	1	2	3
回数	0.3243	0.3736	0.7849	0.8119
サイズ	0.9183	0.9394	0.5822	0.5438
(近似曲線)	0	1	2	3
a	2.000	1.000	5.000	4.000
n	0.8893	0.9365	0.6438	0.7076
R^2	0.8771	0.8399	0.7001	0.5685

表 3: 実験データの分析結果 (比較)

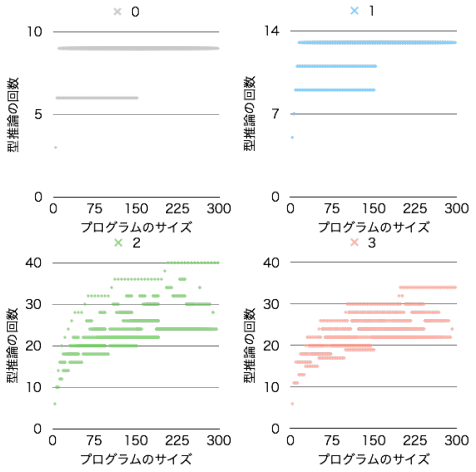

図 9: サイズと回数の散布図 (比較)

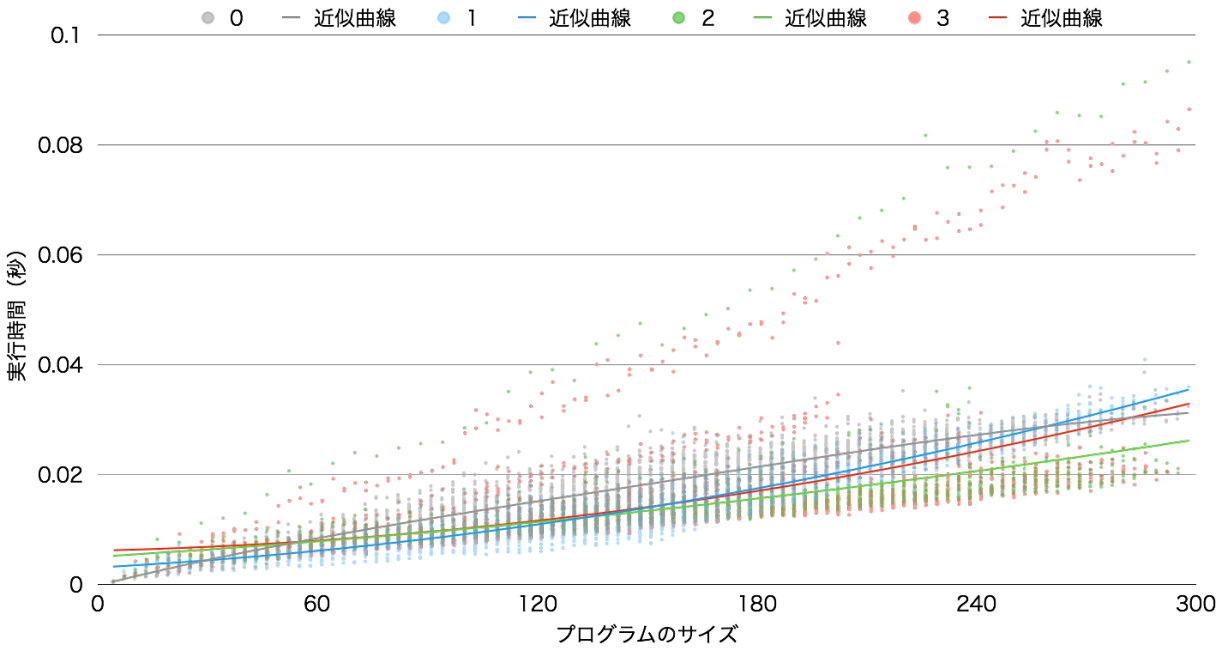

図 10: サイズと時間の散布図 (比較)

型推論の回数と実行時間の相関が強いため，ノードの偏りによって型推論の回数に大きな差が出ると考えられる．

3.1.3 リスト構造

リスト構造は，空リストもしくは先頭の要素と末尾からなる再帰的に定義された多相型として定義されている．そのため，ノードの偏りが大きい上に，全要素の型が同一でなければならないことから，型推論に要する時間が長くなると考えられる．実験の対象となるプログラムは

```
[true; 1], [1; true], [true; 1; 1], [1; true; 1], [1; 1; true], ...
```

のように，要素のひとつの型を *bool*，それ以外を *int* とした式を生成し，要素の数と型を *bool* とする要素の位置を様々な組み合わせで生成した．OCaml の定義上，木の深さは要素数の 2 倍である．実験データの分析結果を表 4 に，プログラムのサイズと型推論の実行回数の散布図を図 11 に，プログラムのサイズと実行時間の散布図を図 12 に示した．なお，型推論の回数の最小値は，*slicer*0 が 5，*slicer*1 と *slicer*2 が 6，*slicer*3 が 7 であった．

(効率)	0	1	2	3
時間	(1.000)	1.031	6.058	5.960
回数	(1.000)	0.9987	4.959	4.798
(相関係数)	0	1	2	3
回数	0.9618	0.9609	0.5318	0.6066
サイズ	0.9618	0.9609	0.5818	0.6182
(近似曲線)	0	1	2	3
a	0.1463	0.1107	0.7051	0.6168
n	1.831	1.887	1.123	1.156
R^2	0.9858	0.9841	0.5031	0.5225

表 4: 実験データの分析結果 (リスト)

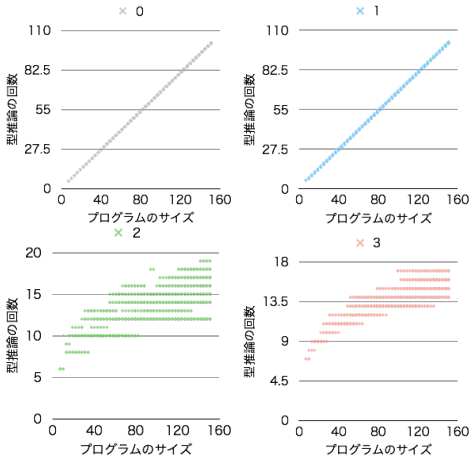

図 11: サイズと回数の散布図 (リスト)

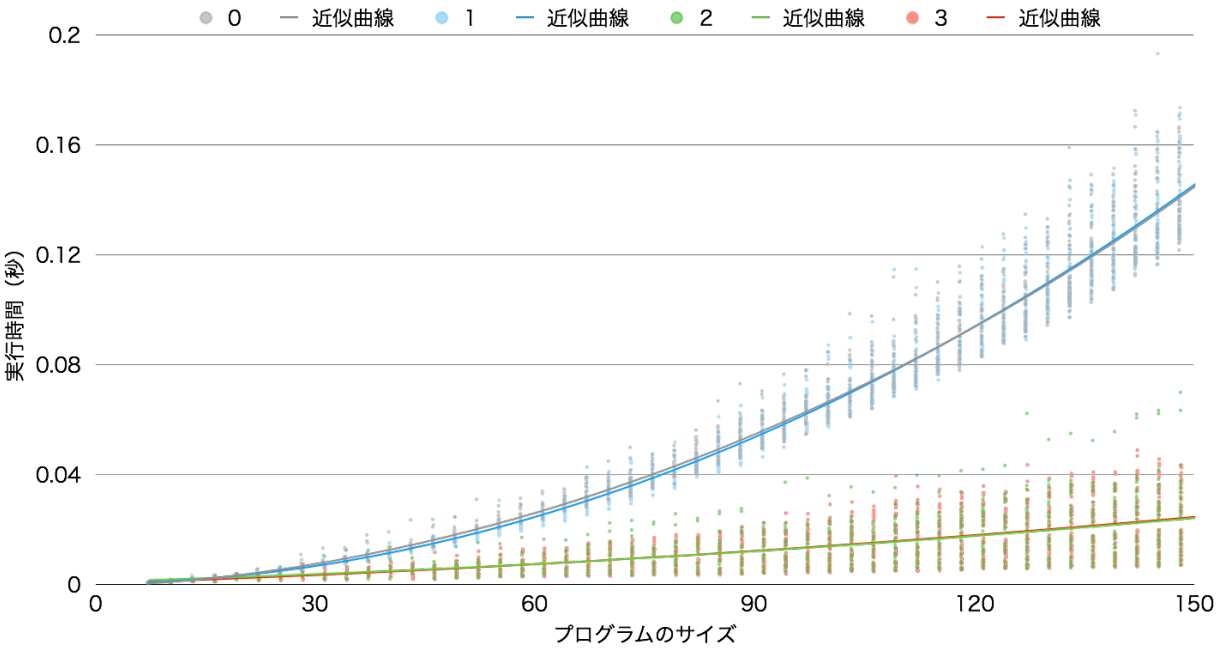

図 12: プログラムのサイズと実行時間の散布図 (リスト)

リスト構造を中心としたプログラムでは，*slicer*2 や *slicer*3 のように全体の半分程度の大きさのノードを選ぶ方法が効率的であった．この傾向は算術演算より強く現れた．*slicer*0 と *slicer*1 では，型推論の回数のばらつきがほとんど見られず，プログラムのサイズが型推論の回数に比例した．計算量は近似式より，計算量は *slicer*0 と *slicer*1 が $O(n^2)$，*slicer*2 と *slicer*3 が $O(n)$ と考えられる．散布図を見ると，全体的にデータのばらつきが小さく，*slicer*0 と *slicer*1，*slicer*2 と *slicer*3 のそれぞれで近似曲線が非常に近い．*slicer*1 の実行時間は *slicer*0 とほとんど変わらないことから，全体の半分程度の大きさのノードを選ぶ方法が効果が特に高く現れている．*slicer*0 と *slicer*1 は，型推論の回数，実行時間，プログラムのサイズ全てに非常に強い相関があった．抽象化されるノードの候補が直下の子ノードに限られているため，型エラーを起こす式が非常に深い位置にあると，型推論の回数が非常に多くなっていることが原因であると考えられる．

3.2 「関数型言語」のプログラム

「関数型言語」で作成するプログラムは，ごくごく簡単なサイズが小さいものから複雑でサイズが大きいものまで幅広く存在する．これらを対象とする実験では，実行時間の効率のみに着目した．プログラムのサイズ毎に提案手法の効率は，該当する全てのプログラム，効率が 1 未満になるプログラム，効率が 1 以上になるプログラムに分類し，各々で平均を求めた．実験データの分析結果を表 5 に示した．なお，実行時間の平均は *slicer*0 が 62.05 ミリ秒，*slicer*1 が 62.60 ミリ秒，*slicer*2 が 63.93 ミリ秒，*slicer*3 が 62.49 ミリ秒であった．

(サイズ)	1			2			3		
0〜19	1.205	0.7516	1.533	1.138	0.6705	1.869	1.487	0.6657	1.487
20〜39	1.202	0.7447	1.471	1.139	0.6490	1.851	1.638	0.6580	1.638
40〜59	1.249	0.7967	1.410	1.162	0.7069	1.614	1.516	0.7494	1.516
60〜79	1.209	0.8572	1.404	1.157	0.8010	1.598	1.482	0.8187	1.482
80〜99	1.165	0.8649	1.273	1.144	0.7549	1.530	1.410	0.7676	1.410
100〜119	1.145	0.8593	1.292	1.132	0.7841	1.515	1.376	0.7955	1.376
120〜139	1.072	0.8803	1.198	1.003	0.7684	1.248	1.201	0.7861	1.201
140〜159	1.052	0.8757	1.190	0.978	0.7854	1.291	1.266	0.7804	1.264
160〜179	1.010	0.8457	1.162	0.950	0.8210	1.197	1.195	0.8155	1.195
180〜199	1.008	0.8612	1.128	1.058	0.7873	1.304	1.193	0.8196	1.193
(200 以上の内訳は省略)									
全体	1.134	0.8740	1.2793	1.095	0.8034	1.598	1.058	0.8076	1.460
効率が上がったデータの割合	0.5462 (54.62%)			0.4180 (41.80%)			0.4437 (44.37%)		

表 5: 実験データの分析結果（「関数型言語」のプログラム）

「関数型言語」のプログラムでは，*slicer*1 に比べて，*slicer*2 と *slicer*3 は効率が 1 未満になるプログラムではより低く，1 以上になるプログラムではより高くなった．加えて，プログラムのサイズが大きいほど 1 以上になるプログラムの効率の平均が低くなる傾向があったため提案手法は有用であり，いずれも効率が上がったデータの割合は半数程度かつ全体では効率が低下しているが，各手法の実行時間の差は小さくさほど問題にならないといえる．明らかな実用性の向上が見られなかった理由として

- プログラムのサイズは大きいが，式あたりのサイズは小さい
- 効率が上がりにくい構文（比較演算など）が含まれる式が多い
- structure の要素数が多いプログラムでは，型エラーに起因しない式が多い

などが考えられる．

3.3 サイズが大きいプログラム

「関数型言語」で実用性の向上が見られなかった理由のうち，分析結果からプログラムのサイズが最も有力であると仮定した．そこで，サイズが大きいプログラムとして，3 種類のプログラム comp.ml，model.ml，infer.ml を用意した[1]．これらはサイズが 200 以上の式を含み，プログラム全体のサイズは 700 以上である．用意したプログラム中のから部分式をランダムに選んで編集し，意

[1] https://github.com/OcamlSlice/test-suite にて該当ファイルを公開している

図的に型エラーを発生させたプログラムを 10 パターンずつ用意した．model.ml のみ他の 2 プログラムと比べてサイズは平均は小さく，分散が非常は大きくなっているが，これは複数の式からなるプログラムで，型エラーが発生する式以降は全て切り捨てられるためである．comp.ml は model.ml と比較的似た構造となっているが，編集する箇所を特定の関数から選んでいる．infer.ml は局所関数定義を繰り返しているため，プログラム中には非常に大きい式がひとつだけ定義されている．実験データの分析結果を表 6 に示した．

(実行時間)	0	1	2	3
comp	379.5	283.6	286.2	268.5
model	186.8	78.40	96.70	100.7
infer	1168	1146	583.9	529.5
(型推論の回数)	0	1	2	3
comp	41.40	34.20	34.90	31.50
model	53.00	29.50	33.30	35.40
infer	87.20	76.10	28.70	29.30

(a) 平均値

(実行時間)	1	2	3
comp	1.208	1.215	1.303
model	1.901	1.689	1.593
infer	1.248	4.093	3.272
(型推論の回数)	1	2	3
comp	1.373	1.359	1.414
model	2.106	1.653	1.637
infer	1.080	2.449	2.822

(b) 効率

表 6: 実験データの分析結果（サイズが大きいのプログラム）

全ての場合に提案手法によって実用性が向上した．comp は *slicer*3 による効果がやや高く，model は *slicer*1 による効果が最も高く，infer は *slicer*2 と *slicer*3 による効率が非常に高くなった．よって，プログラム中の型エラースライスに必要な式の割合が高く，型エラースライスの原因となり得る式のサイズが大きい場合に，提案手法の実用性が大幅に向上するといえる．

4 関連研究

Chitil[1] は合成的な型推論によって最汎型木を生成し，対話的な型エラーデバッガを作成した．対馬らの型エラーデバッガ [7] では，コンパイラの型推論によって最汎型木を生成している．最汎型木の探索はアルゴリズミックデバッギングによって行うが，この手法は Shapiro によって提案された [4]．型エラースライスの研究では，Haack[2] らは，型制約を解決して型エラースライスを生成する手法を提案した．また Schilling[3] は本研究と同様のコンパイラの型推論器を使用する手法を提案した．これらふたつは実際のプログラムでの実験を行っておらず，スライシングの効率に関しての改良は行っていない．

既存研究では，単純な型エラースライス手法 [6, 9] を実用的に使用可能な範囲で実装し，型エラースライスの有用性を確認している．この問題点は大きいプログラムでの実行速度であった．本論文の提案手法は既存研究の手法を元に実装した．既存研究のうち重み付き型エラースライス [6] は，型エラースライスに付加情報を与える点が類似しているが，その目的は大きく異なる．対馬らは型エラーデバッグの際の指針として，付加情報を利用している．型が衝突した回数を明示することによって，あらかじめ型エラーの原因である可能性が高い箇所が予想できるため，ヒューリスティックに良いスライス，およびスライス内での重要度の情報を得ることを目的としている．それに対して，本論文ではプログラム分割の基準値として，効率的に型エラースライスを求めるために利用している．すなわち，型推論の無駄を減らすことによる型エラースライシングの高速化を目的としている．

5　まとめと今後の課題

本論文では複数の型エラースライス手法を提案し，効率を重視して各々を評価した．基準となるノードを選択し，そのノード以外を抽象化する場合とそのノードのみを抽象化する場合とを切り替えるようなアルゴリズムを，基準となるノードの候補の範囲で 2 通り提案した．一方は既存手法と同様に子ノードのみ，もう一方は全体を対象とする．後者はさらに基準となるノードの探索方法を 2 通り実装した．実験の結果，複数のノードを同時に抽象化することによって，実行速度の向上に効果が得られることを確認した．ノードの偏りが大きいプログラムでは，プログラムのサイズ，実行時間，型推論の回数を統計的に分析し，時間計算量も予想した．型エラーを起こすノードが深い位置にあった場合には，プログラムをおおよそ半分に分けるような部分木を選んで抽象化する手法によって，大幅な実行時間の短縮を期待できることがわかった．しかし，既存研究の手法のほうが高速となる場合もあった．式のサイズが小さく，型エラーに起因する式の割合が高い場合に，提案手法による効果が現れにくいと考えられる．

今後の課題として，まずは提案手法による高速化の効果があまり見られないプログラムに対する詳細な分析が挙げられる．適切な手法を選択することによって，型エラースライサーの効率のさらなる向上が期待できる．より効率的に型エラースライスを求めるために，提案手法による高速化が期待できない場合の条件を詳細に求めたい．加えて，対応する構文の拡張およびにアルゴリズムのさらなる改良を検討したい．

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] C. Haack and 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'03)*, Vol. 50 Issue 1-3, , 2004.

[3] T. Schilling. Constraint free type error slicing. *Proceedings of the 12th international conference on Trends in Functional Programming (TFP'11)*, pp. 1–16, 2012.

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

[5] K. Tsushima and K. Asai. An embedded type debugger. *In Implementation and Application of Functional Languages, Lecture Notes in Computer Science 8241*, pp. 190–206, 2012.

[6] 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.

[7] 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.

[8] 対馬かなえ, 脇川奈穂. 実用的な型エラースライサーに向けた改良と評価. 日本ソフトウェア科学会第 34 回大会 講演論文集, 2017.

[9] 脇川奈穂, 対馬かなえ, 浅井健一. 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. 第 58 回プログラミングシンポジウム 予稿集, 2016.