実用的な型エラースライサーに向けた改良と評価

対馬 かなえ　脇川 奈穂

本論文ではコンパイラの型推論器を用いた効率的な型エラースライス手法を提案し、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の中でひとつの型エラーに関係する部分のみを集めたプログラムであり、デバッグ等で役立つ。今まで型エラースライサーは多く提案されてきたが、効率性や実際の速度についての研究はなされてこなかった。本論文では、型エラースライシングを行う際に、おおよそプログラムを半分にわけるように改良した手法を二種類導入する。加えて、それぞれの手法が人工的に作成したサイズが大きいプログラムに対してどの程度の速度か実験し、結果の考察を行う。

1　はじめに

本論文では、型エラースライス [2] [5] の効率的な求め方を提案する。型エラースライスとは、型エラーのプログラムの中でひとつの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。デバッグ時にプログラマにとって便利であるほか、型エラーデバッガ [1] [4] [6] [8] のデバッグ対象箇所を減らすことができる。

まず型エラースライスとはどのようなものか見てみよう。□ を型制約のないプログラム (例外の発生など) とする。OCaml の型エラーになるプログラム (`fun x → x + 3) true` の型エラースライスは (`fun x → x + □) true` である。□ になった部分の型制約がなくてもプログラムは依然として型エラーになることがわかる。

また、型エラースライスには極小という概念が存在する。極小な型エラースライスでは、スライスに含まれるどの部分も型エラーに関係していることが求められる。上の例も極小な型エラースライスであるが、そのスライスに含まれる部分、`x` や `+` など全ての部分は、全体として型エラーになるために必須である。それは、それらを □ で置き換えて、型推論器にかけると型エラーでなくなってしまうことからわかる。

型エラースライスの求め方はいくつかの手法に分けられる。

(1) プログラムから型制約を抽出し、型エラーになる極小の組み合わせを求め、型制約をプログラムへ戻す

この手法の利点は、型制約さえ抽出してしまえば、既存のソルバ等を使用して、極小な型エラースライスが求められることである。[2] など多くの先行研究ではこの手法が使われている。欠点は正しい型制約を抽出するために、その言語の型と型推論に関しての知識が必要になることである。

(2) プログラムを抽象化し、コンパイラの型推論器を使用して型エラーになるかを判定することで、型エラーになる極小のプログラムを求める

この手法の利点は、型推論器をブラックボックスとして使うことができることである。既存の研究 [3] [5] はこちらにあたる。それぞれの構文に対して一段階抽象化したプログラムを定義するだけで、型推論器を実装することなく、既存の型推論器を利用してスライスを求めることができる。大きい言語を対象として挙動が正しいスライサーを定義するためには、型規則の完全な理解が必要ないこちらの手法の方が容易である。

Practical Type Error Slicing

Kanae Tsushima, 国立情報学研究所, Wakikawa Naho, お茶の水女子大学.

実際、著者らは OCaml を対象とした型エラースライサーの実装を行った [7]。欠点としては、抽象化のたびに型推論を行うため、プログラムの大きさと構造によっては時間がかかることである。

これまでの研究 [2] [3] [5] では型エラースライシングの手法の提案や正しさについては検討されてきたが、効率性に関しては研究が行われてこなかった。よって本研究では、既存研究 [5] [7] のスライサーを改良し、スライシングの効率化について検討していく。

2 節では AST の一番上のノードから、順にスライシングを行うスライシング手法 [5] を簡単に振り返る。3 節では、それを改良し、AST の一番上のノードの中で、そのノードのサイズの半分に最も近いものからスライシングを行う手法を導入する。4 節では、より改良を行い、AST 全体の中で全体のサイズの半分に最も近いものからスライシングを行う。5 節では、2 節から 4 節のそれぞれの手法を、人工的に作成したノード数の多いプログラムでどの程度時間がかかるかの実験とその考察を行う。

2 単純に AST の上の方から順に探索するスライサー

本節では、単純なスライサーを簡単に説明する。型エラースライスの厳密な定義、アルゴリズム中の不変条件、アルゴリズムで導出されるものが極小なスライスであることの証明などは論文 [5] を参照されたい。

対象とする言語とその型を図 1 に示した。対象とする言語は定数、変数、ラムダ式、関数適用、組であり、単純型付ラムダ計算に定数と組を追加したものである。図 2 に示したスライス用の構文では、上に加えて □ が追加される。これはスライスで抽象化された式を示す、型制約をもたないプログラムである。AST のある葉が型エラーに関係ない場合には、`(1 + 2.0)` の型エラースライス `(□ + 2.0)` のように葉が □ で置き換えられる。ノード自体が型エラーに関係ないが、その子ノードは関係がある場合には `(x = 3, x = 1.0)` の型エラースライス`@ □ (x = 3) (x = 1.0)` のように、ノード自体が関数部分が □ の関数適用で置き換えられる。

本研究のスライシング手法では、既存の型推論器を使用した関数 `illtyped?` を使用する。これは型推論器を用いて型がつく時には false を、型がつかない時には true を返す関数である。

単純なスライサーの定義を図 3 に示した。*abst_one* は受け取ったプログラムを (□ で置き換えて) 一段階抽象化したもののリストを返す関数である。*choose* は抽象化されたものの中から一つを選ぶ関数である (ただし、本節の *choose* は単純に一番先頭のものを選ぶ)。スライシングのメイン関数 *slicing*1 では以下のようにスライシングを行う。*abst_one* を使用して、一段階抽象化したもののリストを得る。そのリストと、その時のコンテキスト *cxt* および、一段階の抽象化したものに全て型がついた場合のサンクで *search* を呼び出す。*search* では *choose* を使用して、一段階抽象化したものから一つ選ぶ。もしそれに型がつかないならば、(*cxt e*) は型エラーになるのに十分であるので、再び *slicing*1 を呼ぶ。もし型がつくならば、*search* に再帰して残りのリストから他の可能性を探す。他の可能性がない場合には、あらかじめ渡されていた三つ目の引数のサンクを実行する。

この手法では、AST の一番上のノードの子ノードから順にひとつずつ抽象化を行っていく。よってこの手法で時間がかかる例は、バランスが偏った深い木で AST の一番下の方に型エラーの原因が含まれているような場合である。実はそのような例は OCaml では簡単に実現できる。OCaml では大きいリストはコンストラクタの大きい入れ子になる。よって要素がたくさん入ったリストを作成し、最後のみ型が違う要素を入れれば良い。本節の手法を要素数が 200 のリスト (`[1;2;3;...;199;true]`) で試した場合、スライシングに 13.15 秒かかった。

3 AST の上の方から探索するが、サイズが全体の半分に近いものを優先するスライサー

本節では、前節の単純なスライサーを少し改良する。基本的にスライシング前にはどの部分が型エラーに関係があるか分からないため、要素数が半分の状態で型がつくかを試すのが効率的であると考えられる。そこで、前節と同じように AST の一番上のノードか

$(s : int)$	:=	n	(プログラムのサイズ)
$(M : term)$	:=	c^s	(定数)
	|	x^s	(変数)
	|	$\lambda^s x.M$	(関数抽象)
	|	$@^s M \ldots M$	(関数適用)
	|	$let^s\ x = M\ in\ M$	(let 文)
	|	$(M, \ldots, M)^s$	(組)
$(\tau : type)$	:=	α	(型変数)
	|	$int, float, \ldots$	(型定数)
	|	$\tau \to \tau$	(関数型)
	|	$\tau * \ldots * \tau$	(組の型)

図 1　ラムダ計算の構文と型

$(S : slice)$	:=	c^s	(定数)
	|	x^s	(変数)
	|	$\lambda^s x.S$	(関数抽象)
	|	$@^s S \ldots S$	(関数適用)
	|	$let^s\ x = S\ in\ S$	(let 文)
	|	$(S, \ldots, S)^s$	(組)
	|	$\Box$	(ホール)

図 2　スライスの構文

ら探索するが、子ノードの中でもサイズが全体の半分に近いものを優先してみる。

定義を図 4 に示した。プログラムのサイズを求める関数 *size* はスライシング開始時に使用され、プログラムの各ノードにサイズ情報を付与する。*search* の 4 つめの引数、プログラムサイズ n の情報は *choose* でノードを選ぶ際に使用される。*choose* は候補のスライスのリストとサイズ n を受け取り、サイズが最も n/2 に近いスライスとそれ以外のリストを返す。

この手法では、AST の一番上のノードの子ノードの中から、最もそのサイズの半分に近いものを選ぶ。この手法では、バランスが偏った木でも、小さいものを一つ一つ見ていくのではなく、存在するならば、半分くらいに抽象化したものから見ていくことが可能となる。前節の例 `[1;2;3;...;199;true]` (要素が 200 の場合) では 7.90 秒と、前節の単純なスライシングに比べて半分程度の時間となった。これは前節のものは `1::□` を型推論してから `□::...` を行っていたのが、`□::...` を直接行うようになったからであると考えられる。

4　AST の中で最もサイズが全体の半分に近いものを優先するスライサー

本節では、プログラム全体の大きさの半分に近いノードを選ぶスライサーを導入する。前節では対象となる AST の直下の子ノードのみを対象としたが、本節では全てのノードの中で、サイズが全体の半分に近いものを選ぶ。サイズが全体の半分に近いものを選んだのち、型推論を行い、どの部分が必要なのか判定する。対象となる部分 *exp* およびそのコンテキスト *cxt* が必要・不要で 3 パターンが存在する †1。

(1) 対象となる部分 *exp* を □ にしても型がつかない
→ *exp* が不要、$(cxt\ \Box)$ の型エラースライスを求めれば良い

(2) コンテキストを抽象化しても型がつかない
→ *cxt* が不要。

(3) 対象となる部分、コンテキストの両方が必要である
→ *cxt* と *exp* の両方が必要。*cxt* を前提とした上で、*exp* の探索を行う

このような処理を行うために、ふたつのコンテキストを用いる。ひとつはそのままのコンテキスト *exp_cxt* (上の *cxt* と同じもの)、もうひとつはコンテキストの中でもすでにスライスされたものと (スライスの対象ではない) 束縛だけを持つコンテキスト *slice_cxt* である。

†1　$(cxt\ exp)$ が型エラーになるプログラムのみがスライサーに渡されるため、両方とも不要なパターンは存在しない。

$$
\begin{array}{lcl}
abst\_one & : & slice \to slice\ list \\
abst\_one[\![S]\!] & = & ERROR\ when\ S = c^l, v^l \quad (*\ \text{この引数では呼ばれない}\ *) \\
abst\_one[\![\lambda^l x.S]\!] & = & [\lambda^l x.\Box] \setminus (\lambda^l x.S) \\
abst\_one[\![@^l S_1 \ldots S_n]\!] & = & [@\ \Box S_1 \ldots S_n; @^l \Box \ldots S_n; @^l S_1 \ldots \Box] \setminus (@^l S_1 \ldots S_n) \\
abst\_one[\![@\ \Box S_1 \ldots S_n]\!] & = & [@\ \Box S_1 \ldots \Box; \ldots; @\ \Box\Box \ldots S_n] \setminus (@\ \Box S_1 \ldots S_2) \\
abst\_one[\![(S_1, \ldots, S_n)^l]\!] & = & [(\Box, \ldots, S_n)^l; \ldots; (S_1, \ldots, \Box)^l; @\ \Box S_1 \ldots S_n] \setminus (S_1, \ldots, S_n)^l \\
 & & \\
choose & : & slice\ list \to slice * (slice\ list) \\
choose(fst :: rest) & = & fst \\
 & & \\
search & : & (slice\ list * (slice \to slice) * (unit \to slice)) \to slice \\
search([], cxt, f) & = & f() \\
search(lst, cxt, f) & = & let\ (e, rest) = choose\ lst\ in \\
 & & if(\texttt{ill\_typed?}(cxt\ e)) \\
 & & then\ slicing1[\![(e, cxt)]\!] \\
 & & else\ search(rest, cxt, f) \\
 & & \\
slicing1 & : & (slice * (slice \to slice)) \to slice \\
slicing1[\![(\Box, cxt)]\!] & = & ERROR \quad (*\ \text{この引数では呼ばれない}\ *) \\
slicing1[\![(v, cxt)]\!] & = & v \\
slicing1[\![(c, cxt)]\!] & = & c \\
slicing1[\![(\lambda x.S, cxt)]\!] & = & search(abst\_one[\![\lambda x.S]\!], cxt, (fun\ () \to \\
 & & \qquad \lambda x.(slicing1'[\![(S, (fun\ y \to cxt(\lambda x.y)))]\!])) \\
slicing1[\![(@S_1 S_2 \ldots S_n, cxt)]\!] & = & search(abst\_one[\![@S_1 S_2 \ldots S_n]\!], cxt, (fun\ () \to \\
 & & let\ S_1' = slicing1'[\![(S_1, (fun\ x \to cxt(@x S_2 \ldots S_n)))]\!]\ in \\
 & & let\ S_2' = slicing1'[\![(S_2, (fun\ x \to cxt(@S_1' x \ldots S_n)))]\!]\ in\ \ldots \\
 & & let\ S_n' = slicing1'[\![(S_n, (fun\ x \to cxt(@S_1' S_2' \ldots x)))]\!]\ in \\
 & & (@S_1' S_2' \ldots S_n')) \\
slicing1[\![((S_1, S_2, \ldots, S_n), cxt)]\!] & = & search(abst\_one[\![(S_1, S_2, \ldots, S_n)]\!], cxt, (fun\ () \to \\
 & & let\ S_1' = slicing1'[\![(S_1, (fun\ y \to cxt(y, S_2, \ldots, S_n)))]\!]\ in \\
 & & let\ S_2' = slicing1'[\![(S_2, (fun\ y \to cxt(S_1', y, \ldots, S_n)))]\!]\ in\ \ldots \\
 & & let\ S_n' = slicing1'[\![(S_n, (fun\ y \to cxt(S_1', S_2, \ldots, y)))]\!]\ in \\
 & & (S_1', S_2', \ldots, S_n')) \\
slicing1'[\![(S, cxt)]\!] & = & if\ S = \Box\ then\ \Box\ else\ slicing1[\![(S, cxt)]\!]
\end{array}
$$

図 3　単純な型エラースライサ

具体例として、型エラーのプログラム `(fun x → x + (x + x * x)) 2.0` (サイズ 13) を考える。まず最もサイズが全体の半分に近いのは exp = `x + x * x` (サイズ 6) である。その時のコンテキストは以下のようになる。

exp_cxt = `(λ k.(fun x → x + k)) 2.0`

$size$: $slice \to int$

$size[\![\square]\!]$ = $\square^0$

$size[\![c]\!]$ = c^1

$size[\![x]\!]$ = x^1

$size[\![\lambda x.M]\!]$ = $let\ M^s = size[\![M]\!]\ in\ (\lambda x.M^s)^{(s+1)}$

$size[\![@M_1...M_n]\!]$ = $let\ M_1^{s1} = size[\![M_1]\!]\ in...$
$let\ M_n^{s2} = size[\![M_n]\!]\ in$
$@^{(s1+...+s2+1)} M_1^{s1}...M_n^{s2}$

$size[\![(M_1,...,M_n)]\!]$ = $let\ M_1^{s1} = size[\![M_1]\!]\ in...$
$let\ M_n^{s2} = size[\![M_n]\!]\ in$
$(M_1^{s1},...,M_n^{sn})^{(s1+...+sn+1)}$

$choose$: $slice\ list * int \to slice * (slice\ list)$

$choose(lst, n)$ = サイズが最も n/2 に近いスライスとそれ以外のリストを返す

$search$: $(slice\ list * (slice \to slice) * (unit \to slice) * int) \to slice$

$search([], cxt, f, n)$ = $f()$

$search(lst, cxt, f, n)$ = $let\ (e^n, rest) = choose\ lst\ n\ in$
$if(\texttt{ill\_typed?}(cxt\ e))$
$then\ slicing2[\![(size[\![e]\!], cxt)]\!]$
$else\ search(rest, cxt, f, n)$

$slicing2$: $(slice * (slice \to slice) * int) \to slice$
$slicing1$ の定義中の $slicing1$ を $slicing2$ に置き換えたもの
$slicing2(M^n, cxt)$ のサイズ n を $search$ を呼ぶ際の 4 つめの引数として渡す

図 4　全体のサイズの半分に近い子ノードを選ぶ型エラースライサ

$slice\_cxt = \lambda$ `k.(fun x → □ □ k)`
これらのコンテキストと式で、先ほど示したパターンを試してみる。exp_cxt □ は型がつかない。そのことから exp の部分は全体を型エラーにするために不要であることがわかる。よって対象のプログラムを `(fun x → x + □) 2.0` (サイズ 6) と変更し、はじめからスライシングを行う。

`(fun x → x + □) 2.0` の最もサイズが全体の半分に近いノードは $exp =$ `x + □` (サイズ 3) である。その時のコンテキストは以下のようになる。

$exp\_cxt = \lambda$ `k. (fun x → k) 2.0`
$slice\_cxt = \lambda$ `k. (fun x → k) □`
これらのコンテキストと式で、先ほど示したパターンを試してみる。$slice\_cxt\ exp$、exp_cxt □ 共に型がつく。よって、exp_cxt と exp の両方が必要であることがわかる。exp_cxt を前提として、exp を対象にスライシングを行う。

対象のプログラムが `x + □` と変更された (前提は `(λ k. (fun x → k) 2.0)`)。この時サイズが全体の半分に近いと選ばれるノードを $exp =$ `+` と仮定する。†2 するとその時のコンテキストは以下のようになる。

$exp\_cxt =$ `(λ k. (fun x. k x □) 2.0)`

†2　実際には `x` も同じ大きさなので、どちらが選ばれるかは実装に依存する。

slice_cxt = (λ k. (fun x → k □ □) 2.0)

(λ k. (fun x → k) 2.0) が前提となっているため、*slice_cxt* にまだスライス済みではない 2.0 が含まれていることに注意してほしい。*exp_cxt* □ および *slice_cxt exp* は型がつく。よって、+ は型エラーになるために必須であることがわかる。定数 + はスライスに必要であると確定したため、$+^0$ のようにサイズを 0 にする。

対象のプログラムが (λ k. (fun x. x $+^0$ □) 2.0) と変更された。全体のサイズを付け直し、再びはじめからスライシングを行う。上と同じようなプロセスで x および 2.0 も必須であることが確定し、最終結果として、(fun x → x^0 $+^0$ □) 2.0^0 が得られる。

図 5 に定義を示した。関数 *slice*? は、スライス済み (サイズが 0) の部分を残し、それ以外の部分を □ で置き換える関数である。これは *slice_cxt* を作るために使われる。関数 *sublist* は引数の式の、子ノードとその環境ふたつのリストを返す。一つ目の環境 *exp_cxt* は子ノードを e としたとき (*exp_cxt* e) が引数の式になるようなものである。二つ目の環境 *slice_cxt* は *exp_cxt* のうちスライス済みの部分はそのままで、それ以外の部分は抽象化した環境である。関数 *pickup* は一つ目の引数 *exp* の中から、四つ目の引数 n の半分の大きさに近いノードを見つけて返す関数である。同時にその *exp_cxt* と *slice_cxt* も返す。定義中の *biggest* の定義は省略してあるが、リストの中から最も大きい式を選んで返す関数である。

関数 *update* はスライスを受け取り、サイズを再計算して付与したのち、全体がスライス済みならば終了して結果を返し、スライス済みでなければ再び *slicing3* を呼び出す関数である。*id* は初期の環境で、$fun\ x \to x$ で表される。関数 *slicing3* はスライサーのメイン関数である。パターン分けを行い、不要なコンテキストおよび式を置き換えて、スライシングを進める。*update* を呼び出している部分は、葉まで到達したので再び AST の全体をスライシングしている。*slicing3* に再帰している部分は、*exp_cxt* もしくは *hole_cxt* を前提にして、子ノードのスライシングを行っている。

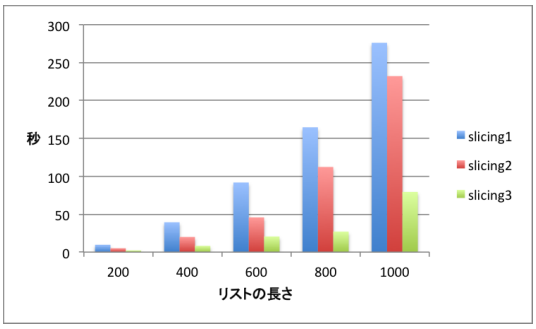

図 6　リストの長さに応じた各スライシングの実行時間 (エラーの原因が最下部にある場合)

2 節の例 [1;2;3;...;199;true] では、2.92 秒と、単純なスライシングに比べて 1/4、前節のスライシングに比べて 1/2 程度の時間となった。半数ずつに減らしていった結果、高速化できていることがわかる。

5　実験と評価

本節では、2 節から 3 節の手法がどの程度効率的に実行できているか、人工的に作成したサイズの大きいプログラムを用いて評価を行う。

まず、前節までで使用していたリストの例 ([1;2;3;...;199;true]) で長さを変更した実行結果を図 6 に示した。図からわかるように、どの手法でもリストの長さが長くなるほど、実行時間がかかる。slicing1 の実行時間を見てみると、単純な比例ではない。長さを n とした時の実行時間 t はおおよそ $t = 0.00025 * n^2$ となる。これはリストの長さの回数分、リストの長さのプログラムの型推論が行われていることに起因すると考えられる。slicing2 の実行時間を見てみると、長さが 600 までは slicing1 のおおよそ半分である。これは (n :: □) ではなく (□:: ...) がはじめに実行されていることに起因すると考えられる。slicing3 の実行時間を見てみると、長さを n とした時の実行時間 t はおおよそ $t = 0.00005 * n^2$ とおよそ 1/5 になっている。

上の例は AST の右側への偏りがある例であるが、左側への偏りも同様の傾向が見られる。足し算の例 (true + 1 + 2 + ... + 199) の実行結果を図 7 に示した。前の例に比べて、slicing2 および slicing3

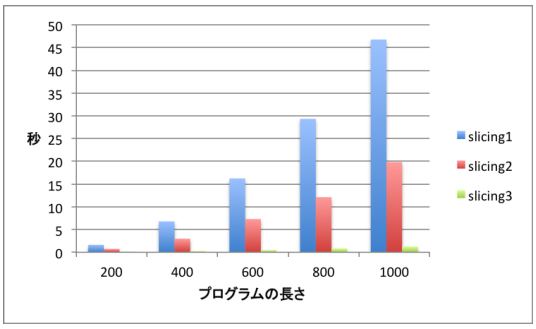

図 7　足し算の長さに応じた各スライシングの実行時間
(エラーの原因が最下部にある場合)

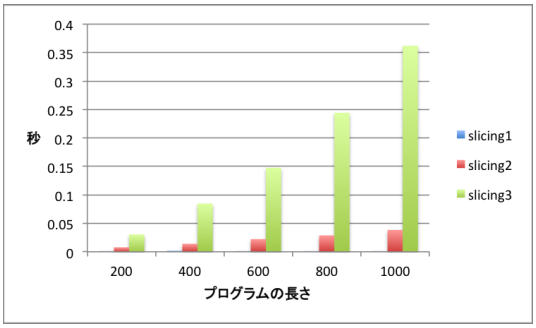

図 8　足し算の長さに応じた各スライシングの実行時間
(エラーの原因が最上部にある場合)

の上がり幅が抑えられている原因はコンストラクタと関数適用の違いかと思われるが、現在さらなる実験で確認中である。

これらの例は AST の最も下の部分に型エラーが含まれている場合で、slicing3 に有利な例であった。そのため、最も不利な場合、最も上部に型エラーの原因が含まれている場合 (`1 + 2 + .... + 199 + true`) を考える。その際の実行結果を図 8 に示した。slicing1 と比較して slicing3 は遅いことがわかる。これは slicing2 や slicing3 では様々な処理を行っているほか、この例では非効率なスライシング順序を選んでいることによる。一方で、一つ目の例や二つ目の例の slicing1 に比べて現実的な時間で抑えられている。

これまでの三つの例は、AST に偏りがある場合であった。よって次には平衡木を考える。要素 512 の足し算の完全な平衡木で、一つだけ bool の値が含まれている例を考える。その実行結果は以下のようになる。

slicing1: 0.048731 (sec)
slicing2: 0.061794 (sec)
slicing3: 0.013203 (sec)

この例では slicing3 が最も早いが、エラーの原因である bool の値を入れる場所を変更すると、それぞれのスライシング手法で 0.02 〜 0.09 程度で変化し、slicing2 が最も早いこともあった。

また、要素のほとんどがエラーの原因であるような例 `fun x -> fun y -> fun z -> fun a -> fun m -> fun n -> fun k -> fun l -> (x = y, y = z, z = a, a = m, m = n, n = k, k = l, x = 2.0, l = 3)` では、

slicing1: 0.110034 (sec)
slicing2: 0.029109 (sec)
slicing3: 0.064413 (sec)

という結果が得られた。

slicing1 では要素を選ぶ操作をしていない分、うまくいった場合のコストが小さいと考えていたが、実験前に想像していたよりもコストは実験からは確認できなかった (図 8 の例は、主にその例では非効率な順序であったことに起因すると考えられる)。多くの例では slicing3 が早いことが多いことがわかる。今後は人工的な例だけではなく、実際のプログラムで実験と評価を行いたい。

6　まとめと今後の課題

本論文では、型エラースライスの効率的な求め方について提案と実験を行った。新たに導入したのは二つの型エラースライサーである。既存研究 [5] での単純な型エラースライサーを改良し、全体のサイズの半分に近い子ノードを選ぼうとする *slicing*2 と、全体をおよそ 1/2 に分けるノードを選ぶ *slicing*3 を導入した。

人工的に作成した大きいリストや足し算の例では、エラーの原因が含まれる場所に依存するが、おおよそ *slicing*3 が高速であることが多いことが確認された。一方で、例によっては *slicing*1 の方が早い場合もあった。

今後の課題は三つ挙げられる。一つ目は実際のコードでの実行時間比較である。実際の型エラーのプログラムの型エラースライシングではどの手法が効率的かの評価を行いたい。二つ目は更なる効率化である。型エラーのメッセージで指摘された箇所をあらかじめスライシングに組み込むなどでさらに効率化できないか確認したい。三つ目は複数の手法をハイブリッドに用いて高速化する方法がないかの検証である。プログラムが小さくなると要素を選ぶコストが存在しない slicing1 の方が早い場合も存在する。要素数が少なくなった場合には、slicing3 から slicing1 に切り替える等でより高速化できないか検討したい。

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

[4] 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).

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

[6] 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).

[7] 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.

[8] Tsushima, K., and Asai, K.. “An Embedded Type Debugger,” *Proceedings of the 24th International Workshop on Implementation of Functional Languages (IFL’12)*, Springer, LNCS 0302–9743, pp. 190–206 (2013).

$$
\begin{array}{lcl}
slice? & : & slice \to slice \\
slice?(S^0) & = & S\ when\ S = v\ or\ c \\
slice?(S^n) & = & \Box\ when\ S = v\ or\ c \\
slice?(\lambda^{l1}x.S) & = & let\ S'^{l2} = slice?(S)\ in \\
 & & if\ l2 = 0\ then\ \lambda^0 x.S^0\ else\ \lambda^{l1}x.S'^{l2} \\
slice?(@^l\ S_1...S_n) & = & let\ S'^{l1}_1 = slice?(S_1)\ in\ ... \\
 & & let\ S'^{ln}_n = slice?(S_n)\ in \\
 & & if\ (l1 = 0\ \wedge ... \wedge ln = 0)\ then\ (@^0\ S'^0_1...S'^0_n) \\
 & & else\ (@^l\ S'^{l1}_1...S'^{ln}_n) \\
slice?((S_1, ..., S_n)) & = & \text{関数適用の場合とほぼ同様なので、省略}
\end{array}
$$

$$
\begin{array}{lcl}
sublist & : & slice \to (slice * (slice \to slice) * (slice \to slice))\ list \\
sublist[\![S]\!] & = & ERROR\ when\ S = v\ or\ c \quad (*\ \text{この引数では呼ばれない}\ *) \\
sublist[\![\lambda^l x.S]\!] & = & [(S, (fun\ k \to \lambda^l x.k), (fun\ k \to \lambda^l x.k))] \\
sublist[\![@^l\ S_1...S_n]\!] & = & [(S_1, (fun\ k \to (@^l\ k...S_n)), (fun\ k \to (@^l\ k...(slice?\ S_n)))); \\
 & & ...; (S_n, (fun\ k \to (@^l\ (slice?\ S_1)...k)))] \\
siblist[\![(S_1, ..., S_n)]\!] & = & \text{関数適用の場合とほぼ同様なので、省略}
\end{array}
$$

$$
\begin{array}{lcl}
pickup & : & slice * (slice \to slice) * (slice \to slice) * int \to (slice * (slice \to slice) * (slice \to slice)) \\
pickup(exp, exp\_cxt, hole\_cxt, n) & & let\ subs = sublist\ exp\ in \\
 & & let\ (exp'^{n'}, exp\_cxt, hole\_cxt) = biggest\ subs\ in \\
 & & if\ n/2 > n'\ then\ (exp'^{n'}, exp\_cxt\ exp\_cxt', hole\_cxt\ hole\_cxt') \\
 & & else\ pickup\ (exp'^{n'}, exp\_cxt\ exp\_cxt', hole\_cxt\ hole\_cxt')
\end{array}
$$

$$
\begin{array}{lcl}
update & : & slice \to slice \\
udpate(exp) & = & let\ exp'^n = size[\![exp]\!]\ in \\
 & & if\ n = 0\ then\ exp' \\
 & & else\ slicing3[\![(exp'^n, id, id)]\!]
\end{array}
$$

$$
\begin{array}{lcl}
slicing3 & : & slice * (slice \to slice) * (slice \to slice) * int \to slice \\
slicing3[\![(\Box, exp\_cxt, slice\_cxt, n)]\!] & = & ERROR\ (*\ \text{この引数では呼ばれない}\ *) \\
slicing3[\![(S, exp\_cxt, slice\_cxt, n)]\!] & = & if\ illtyped?\ (hole\_cxt\ exp)\ then\ update\ (hole\_cxt\ e^0) \\
\quad when\ S = v\ or\ c & & else\ if\ illtyped?\ (exp\_cxt\ \Box)\ then\ update\ (exp\_cxt\ \Box) \\
 & & else\ update\ (exp\_cxt\ e^0) \\
slicing3[\![(S, exp\_cxt, slice\_cxt, n)]\!] & = & if\ illtyped?\ (hole\_cxt\ exp) \\
 & & then\ let(exp', exp\_cxt', hole\_cxt') = pickup\ (exp, id, id, n)\ in \\
 & & \quad slicing3[\![exp', (hole\_cxt\ exp\_cxt'), (hole\_cxt\ hole\_cxt')]\!] \\
 & & else\ if\ illtyped?\ (exp\_cxt\ \Box)\ then\ update\ (exp\_cxt\ \Box) \\
 & & \quad else\ let(exp', exp\_cxt', hole\_cxt') = pickup\ (exp, id, id, n)\ in \\
 & & \quad\quad slicing3[\![(exp',\ (exp\_cxt\ exp\_cxt'),\ (exp\_cxt\ hole\_cxt'))]\!]
\end{array}
$$

図 5　全体で半分に近いノードを選ぶスライサー