多段階計算に基づくコンパイル時テンソル形状検査の DNN プログラムへの適用

諏訪 敬之 *1, *2　五十嵐 淳 *1
*1: 京都大学　*2: Imiron

PPL 2026 @ 高松
2026 年 3 月 9–12 日

github.com/gfngfn/Horsea

多段階計算 [Davies 1996] [Taha+ 1997] に基づき，**テンソル（行列の一般化）の形状に関する整合性をコンパイル時計算で保証し，形状起因の失敗が実行時に決して生じない体系を提案**（ECOOP 2026 で発表予定）

この手法が成立する前提： テンソルを扱う典型的なプログラムは，実際に計算を行なうよりも前に引数や結果のテンソルの形状が抽出できるような演算のみを使って書ける

- 例： 行列積 AB は，A と B の寸法が事前にわかっていれば結果の寸法も事前にわかり，かつ（A の列数）＝（B の行数）も事前に保証可能
- 計算結果の形状が事前にはわからない演算 （`List.filter` のように特定の列だけ行列から抜き出すなど） は，大抵使わずに済むはず

➡ **Q1： この前提は本当に成り立つか？**

➡ A1： ***ocaml-torch*** のリポジトリにある数十〜数百行程度のプログラム例が提案手法の言語 ***Horsea*** へとひとまず 10 個移植できた．残る十数個ほどの例もおそらく移植可能で，ゆえに **DNN プログラムに関しては成り立つと言えそう**

コア言語はテンソル形状に関する註釈的な引数を数多く要するが，或る種の ***bidirectional type-checking*** [Xie+ 2018] によってその多くをユーザが明記せずとも推論できる手法も提唱

➡ **Q2： どの程度の割合で推論でき，どんな場合に難しいか？**

➡ A2： 数え方によるが，上記の移植例では **9 割方推論できた**．高階函数についての推論は不十分で，単一化などがベースのアルゴリズムが真に必要かも

型検査（＋elaboration）の定式化の概略

- cf. 一般的な依存型検査の規則がやっていることの概略：

$$\frac{\Gamma \vdash M_1 : \mathrm{Tensor}\ N'_{11} \to T \qquad \Gamma \vdash M_2 : \mathrm{Tensor}\ N'_2 \qquad \vDash \forall \Gamma\,.\ N'_2 = N'_{11}}{\Gamma \vdash M_1\ M_2 : T}$$

- 提案手法を模式的に表した規則（$\sim$ と $\langle\rangle$ は unquote/quote）：

$$\frac{\Gamma \vdash M_1 : (\mathrm{Tensor}\ N'_{11} \to T) \rightsquigarrow N_1 \qquad \Gamma \vdash M_2 : \mathrm{Tensor}\ N'_2 \rightsquigarrow N_2}{\Gamma \vdash M_1\ M_2 : T \rightsquigarrow (N_1\ \sim(\mathbf{assert}\ (N'_2 = N'_{11}); \langle N_2 \rangle))}$$

構文

$$
\begin{aligned}
T^{(0)} &::= \{\!|\, x : B \mid M^{(0)} \,|\!\} \mid \mathrm{Tensor}\ [n, \ldots, n] \mid \langle T^{(1)} \rangle \\
&\quad \mid (x : T^{(0)}) \to T^{(0)} \mid \{x : T^{(0)}\} \to T^{(0)} \\
T^{(1)} &::= B \mid T^{(1)} \to T^{(1)} \mid \mathrm{Tensor}\ \%M^{(0)} \\
M^{(0)} &::= c \mid x \mid \lambda x : T^{(0)}.\ M^{(0)} \mid M^{(0)}\ M^{(0)} \mid \lambda\{x : T^{(0)}\}.\ M^{(0)} \mid M^{(0)}\ \{M^{(0)}\} \mid \langle M^{(1)} \rangle \\
M^{(1)} &::= c \mid x \mid \lambda x : T^{(1)}.\ M^{(1)} \mid M^{(1)}\ M^{(1)} \mid \sim M^{(0)}
\end{aligned}
$$

コード型

ステージ 1 の Tensor の引数はステージ 0 の式

implicit param. の抽象

型検査の際に elaboration で ステージ 1 の型の等価性を検査する
ステージ 0 の assertion を挿入： $M^{(0)} ::= \cdots \mid (\!|\, \langle T^{(1)} \rangle \Rightarrow \langle T^{(1)} \rangle \,|\!)$

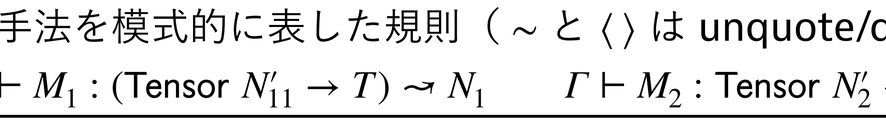
$(\!|\, \langle \mathrm{Tensor}\ \%[1+4] \rangle \Rightarrow \langle \mathrm{Tensor}\ \%[2+3] \rangle \,|\!)$
$\longrightarrow^{*} (\!|\, \langle \mathrm{Tensor}\ \%[5] \rangle \Rightarrow \langle \mathrm{Tensor}\ \%[5] \rangle \,|\!) \longrightarrow \lambda x : \langle \mathrm{Tensor}\ \%[5] \rangle.\ x$

$(\!|\, \langle \mathrm{Tensor}\ \%[5] \rangle \Rightarrow \langle \mathrm{Tensor}\ \%[8, 9] \rangle \,|\!) \longrightarrow$ err

参考： 多段階計算 [Davies 1996] [Taha+ 1997]

$e ::= x \mid e\ e \mid \lambda x\,.\ e \mid \cdots \mid \langle e \rangle \mid \sim e$

- **ブラケット** $\langle e \rangle$ と **エスケープ** $\sim e$ により式が**ステージ**をなす
  - $\langle e \rangle$ がコード断片をつくり，$\sim e$ でコード断片を穴に埋め込む



- 基本的にはステージ 0 の式だけが通常の β 簡約で評価されるが，エスケープ ~ とブラケット ⟨⟩ はステージ 1 で相殺される

- 穴のない $\langle e \rangle$ に到達したらそれがコード生成の終了に相当し，そして完成した e が通常のプログラムとして使われる
- **コード型** ($\tau ::= B \mid \tau \to \tau \mid \cdots \mid \langle \tau \rangle$) により，**型のつくプログラムからは型のつくコードしか生成されないことが保証できる**

表層言語からのステージと implicit param. の復元

束縛時解析 [Jones 1993] と ***Let arguments go first*** [Xie+ 2018] に基づく推論規則で ⟨⟩，~，省略された implicit parameter を復元

```
let repeat_and_add {n : Nat} (u : Vec n) (v : Vec (2 * n)) =
  vadd (vappend u u) v
```

```
repeat_and_add [10, 20, 30] [1, 2, 3, 4, 5, 6]
```

束縛時解析 ＋ 省略引数の位置を特定

Vec %M は Tensor %[M] の構文糖衣

```
let repeat_and_add = λ{n : Nat}. ⟨λu : Vec %n. λv : Vec %(2 * n).
  ~(gen_vadd {□}) (~(gen_vappend {□} {□}) u u) v⟩
```

```
⟨~(repeat_and_add {□}) [10, 20, 30] [1, 2, 3, 4, 5, 6]⟩
```

型検査時に assertion を挿入しつつ，省略引数を推論
（完全性は満たさないが，多くの典型的な例で復元できる）

```
let repeat_and_add = λn : Nat. ⟨λu : Vec %n. λv : Vec %(2 * n).
  ~(gen_vadd (2 * n))
    ~(⦇⟨Vec %(n + n)⟩ ⇒ ⟨Vec %(2 * n)⟩⦈ ⟨~(gen_vappend n n) u u⟩) v⟩
```

```
⟨~(repeat_and_add 3) [10, 20, 30] [1, 2, 3, 4, 5, 6]⟩
```

ocaml-torch の API に対する型つけの例

broadcasting（形状を整合させるキャスト）のサポート：

```
Tensor.gen_add : {s1 : List Nat} → {s2 : {|ν : List Nat | broadcastable s1 ν|}} →
  ⟨Tensor %s1 → Tensor %s2 → Tensor %(broadcast s1 s2)⟩
```

implicit parameter を推論しやすくする工夫：

```
Tensor.gen_cross_entropy_for_logits : {s : {|ν : List Nat | List.length ν = 2|}} →
  ⟨Tensor %s → Tensor %[List.nth 0 s] → Tensor %[]⟩
```

```
Tensor.gen_cross_entropy_for_logits : {m : Nat} → {n : Nat} →
  ⟨Tensor %[m, n] → Tensor %[m] → Tensor %[]⟩
```

- 2 つの型つけは等価だが，implicit parameter の推論に関しては前者の方が簡単．適用時の実引数の型から s に当てはめるべき式が直接得られるため

プログラム移植例： `mnist/linear.hrs`

```
let learning_rate = Tensor.f 1.0 in
let ws = Tensor.zeros (let open MnistHelper in [image_dim, label_count]) in
let bs = Tensor.zeros (let open MnistHelper in [label_count]) in
let model {n : Nat} (xs : Tensor [n, MnistHelper.image_dim]) =
  let open Tensor in mm xs ws + bs
in
range 1 200 |> List.iter (fun(i : Int) ->
  let loss =
    Tensor.cross_entropy_for_logits (model MnistHelper.train_images)
      #targets MnistHelper.train_labels
  in
  Tensor.backward loss;
  Tensor.no_grad (fun(u : Unit) -> let open Tensor in
    ws -= grad ws * learning_rate;  bs -= grad bs * learning_rate);
  Tensor.zero_grad ws;  Tensor.zero_grad bs;
  let got = model MnistHelper.test_images in
  let estimated = Tensor.argmax 1 got in
  let test_accuracy =
    let sum = Tensor.count_equal estimated MnistHelper.test_labels in
    float sum / float (lift_int MnistHelper.num_test_images)
  in
  print_float test_accuracy)
```

移植結果

プログラム	行数	imp. 推論成功/全個数	型註釈数
char_rnn/char_rnn.hrs	118	37 / 39	12
gan/mnist_cgan.hrs	154	55 / 59	5
gan/mnist_dcgan.hrs	195	106 / 112	4
gan/mnist_gan.hrs	142	47 / 51	4
jit/load_and_run.hrs	17	3 / 5	0
min-gpt/mingpt.hrs	321	96 / 108	17
mnist/conv.hrs	72	25 / 28	3
mnist/linear.hrs	29	20 / 20	1
pretrained/finetuning.hrs	89	35 / 45	6
pretrained/predict.hrs	77	5 / 8	0