テンソル形状検査のためにコード生成時アサーションを行うステージつき言語とその Idris 風表層言語

PPL 2025 @ 蒲郡
2025 年 3 月 5–7 日

諏訪 敬之 *1,*2　五十嵐 淳 *1

*1: 京都大学　*2: Imiron

動機

テンソル（行列などの一般化）の絡む計算を行うプログラムでは，
テンソル形状（行列サイズなど）の整合性を実行前に保証したい

しかし ⇒ 一般的な依存型や篩型など，**型つけに証明を要する定式化はしばしば開発プロセスと相性が悪い**

- 手動証明： 将来の拡張やリファクタリングの負担が増加
- 自動証明： 所要時間が予測しづらく，しばしば長い時間を要するため，CI/CD などの開発ワークフローに不適

そこで ⇒ 以下のような定式化ができると嬉しい

- 将来の拡張の負担を不用意に増やしにくい
- 検査の所要時間が十分短い，または少なくとも予測しやすい
- 依然として健全性を満たす整合性の検証ができる
- 不整合検出時に，その原因箇所が “わかりやすい”

Key Idea： 多段階計算 [Davies 1996] [Taha+ 1997] を利用する

- テンソル形状の整合性は**コンパイル時計算（≒マクロ展開）の assertion によって保証**
  - **“validation & instantiation としてのコード生成”**
  - 不整合があれば，コンパイル時計算中に失敗が即座に報告される
- cf. 一般的な依存型検査の規則がやっていることの概略：

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

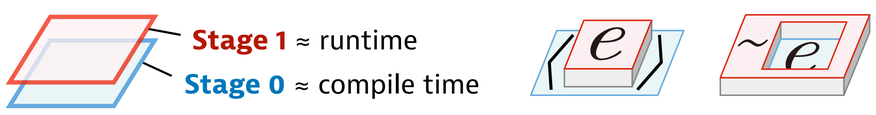
- 提案手法を模式的に表した規則（ $\sim$ と $\langle\rangle$ については後述）：

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

準備： 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 $\sim$ とブラケット $\langle\rangle$ 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** ($\tau ::= \mathsf{Int} \mid \tau \to \tau \mid \cdots \mid \langle \tau \rangle$) 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**

類似する手法との簡単な比較

	将来の拡張	検査時間	実行時の不整合	全称命題の保証
単純型のみ	✔ 負担小	✔ 高速	✖ 起こりうる	N/A
依存型＋手動証明 [Brady 2013][Brady 2021]	☹ 負担大きめ	✔ 高速	✔ 排除できる	✔
篩型＋自動証明 [Rondon+ 2008]	✔ ソルバと闘う必要あり	☹ 長い傾向	✔ 排除できる	✔
篩型 hybrid 方式 [Flanagan+ 2010] [Sekiyama+ 2017] [Hattori+ 2023]	✔ 負担小	✔ タイムアウト可	☹ 自動で証明できなかった箇所については起こりうる	N/A
提案手法	✔ 負担小	✔ 高速	✔ 排除できる	☹ hybrid 拡張しない限り不可

- 通常の手法： テンソル形状についての**ありうる全ての組合せで**プログラムが動作することを静的に（なるべく）保証
- 提案手法： **必要な組合せについて**コード生成し，その過程でいずれも不整合がないことをコンパイル時評価で高速に保証
  - とはいえ **property-based testing** として使用可能 [Claessen & Hughes 2000]

コア言語の構文

コード型　ステージ 0 の依存函数型

Source

$$\begin{aligned}
S^{(0)} &::= \mathsf{Int} \mid \mathsf{Vec}\ n \mid \cdots \mid \langle S^{(1)} \rangle \mid (x : S^{(0)}) \to S^{(0)} \\
S^{(1)} &::= \mathsf{Int} \mid \mathsf{Vec}\ \%M^{(0)} \mid \cdots \mid S^{(1)} \to S^{(1)} \\
M^{(0)} &::= c \mid x \mid \lambda x : S^{(0)}.\ M^{(0)} \mid M^{(0)}\ M^{(0)} \mid \langle M^{(1)} \rangle \\
M^{(1)} &::= c \mid x \mid \lambda x : S^{(1)}.\ M^{(1)} \mid M^{(1)}\ M^{(1)} \mid {\sim}M^{(0)}
\end{aligned}$$

ステージ 1 の Vec の引数はステージ 0 の式

Target

$T^{(i)}$ と $N^{(i)}$ ： $S^{(i)}$ と $M^{(i)}$ の assertion 挿入後版

$$N^{(0)} ::= \cdots(\text{equiv. to } M^{(0)})\cdots \mid (\!|\, \langle T^{(1)} \rangle \Rightarrow \langle T^{(1)} \rangle \,|\!)$$

ステージ 1 の型の等価性を検査するステージ 0 の assertion

$$(\!|\, \langle \mathsf{Vec}\ \%(1+4) \rangle \Rightarrow \langle \mathsf{Vec}\ \%(2+3) \rangle \,|\!)$$
$$\xrightarrow{*} (\!|\, \langle \mathsf{Vec}\ \%5 \rangle \Rightarrow \langle \mathsf{Vec}\ \%5 \rangle \,|\!) \qquad (\!|\, \langle \mathsf{Vec}\ \%5 \rangle \Rightarrow \langle \mathsf{Vec}\ \%8 \rangle \,|\!) \to \mathsf{err}$$
$$\to \lambda x : \langle \mathsf{Vec}\ \%5 \rangle.\ x$$

最小規模の例

$\mathsf{gen\_vadd} : (m : \mathsf{Int}) \to \langle \mathsf{Vec}\ \%m \to \mathsf{Vec}\ \%m \to \mathsf{Vec}\ \%m \rangle$
$\mathsf{gen\_vappend} : (p : \mathsf{Int}) \to (q : \mathsf{Int}) \to \langle \mathsf{Vec}\ \%p \to \mathsf{Vec}\ \%q \to \mathsf{Vec}\ \%(p+q) \rangle$

$$\mathbf{let}\ \mathsf{repeat\_and\_add} = \lambda n : \mathsf{Int}.\ \langle \lambda u : \mathsf{Vec}\ \%n.\ \lambda v : \mathsf{Vec}\ \%(2*n).\ {\sim}(\mathsf{gen\_vadd}\ (2*n))\ ({\sim}(\mathsf{gen\_vappend}\ n\ n)\ u\ u)\ v \rangle$$

型検査時に assertion を挿入

$$\mathbf{let}\ \mathsf{repeat\_and\_add} = \lambda n : \mathsf{Int}.\ \langle \lambda u : \mathsf{Vec}\ \%n.\ \lambda v : \mathsf{Vec}\ \%(2*n).\ {\sim}(\mathsf{gen\_vadd}\ (2*n))\ {\sim}(\!|\, \langle \mathsf{Vec}\ \%(n+n) \rangle \Rightarrow \langle \mathsf{Vec}\ \%(2*n) \rangle \,|\!)\ \langle {\sim}(\mathsf{gen\_vappend}\ n\ n)\ u\ u \rangle)\ v \rangle$$

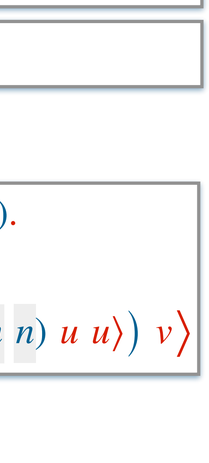
$$\langle {\sim}(\mathsf{repeat\_and\_add}\ 3)\ [10, 20, 30]\ [1, 2, 3, 4, 5, 6] \rangle$$

コンパイル時評価（通常は一瞬で終わり，ここで失敗しうる）

$$\langle (\lambda u : \mathsf{Vec}\ \%3.\ \lambda v : \mathsf{Vec}\ \%6.\ \mathsf{vadd}_6\ (\mathsf{vappend}_{3,3}\ u\ u)\ v)\ [10, 20, 30]\ [1, 2, 3, 4, 5, 6] \rangle$$

実行時の評価（現実的な例ではここが重い処理．不整合なし）

$$[11, 22, 33, 14, 25, 36]$$



表層言語とその束縛時解析・省略引数推論

コア言語ではテンソル形状を指定する多くの引数を与えねばならず，ステージの指定もすべて手動で冗長なため，**省略可能にして補完**

⇒ **束縛時解析** [Jones 1993] と **Let arguments go first** [Xie+ 2018] に基づく推論規則で $\langle\rangle$，$\sim$，省略された引数を復元する

$$\mathbf{let}\ \mathsf{repeat\_and\_add}\ \{n : \mathsf{Int}\}\ (u : \mathsf{Vec}\ n)\ (v : \mathsf{Vec}\ (2*n)) = \mathsf{vadd}\ (\mathsf{vappend}\ u\ u)\ v$$

$$\mathsf{repeat\_and\_add}\ [10, 20, 30]\ [1, 2, 3, 4, 5, 6]$$

束縛時解析 ＋ 省略引数の位置を特定

$$\mathbf{let}\ \mathsf{repeat\_and\_add} = \lambda\{n : \mathsf{Int}\}.\ \langle \lambda u : \mathsf{Vec}\ \%n.\ \lambda v : \mathsf{Vec}\ \%(2*n).\ {\sim}(\mathsf{gen\_vadd}\ \{\square\})\ ({\sim}(\mathsf{gen\_vappend}\ \{\square\}\ \{\square\})\ u\ u)\ v \rangle$$

$$\langle {\sim}(\mathsf{repeat\_and\_add}\ \{\square\})\ [10, 20, 30]\ [1, 2, 3, 4, 5, 6] \rangle$$

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）

$$\mathbf{let}\ \mathsf{repeat\_and\_add} = \lambda n : \mathsf{Int}.\ \langle \lambda u : \mathsf{Vec}\ \%n.\ \lambda v : \mathsf{Vec}\ \%(2*n).\ {\sim}(\mathsf{gen\_vadd}\ (2*n))\ {\sim}(\!|\, \langle \mathsf{Vec}\ \%(n+n) \rangle \Rightarrow \langle \mathsf{Vec}\ \%(2*n) \rangle \,|\!)\ \langle {\sim}(\mathsf{gen\_vappend}\ n\ n)\ u\ u \rangle)\ v \rangle$$

$$\langle {\sim}(\mathsf{repeat\_and\_add}\ 3)\ [10, 20, 30]\ [1, 2, 3, 4, 5, 6] \rangle$$

現在までの貢献と今後の展望

- ✅ コア言語の保存と進行の証明
- ✅ Idris [Brady 2021] 風の表層言語を与え，ステージや一部の省略引数を推論
- ✅ 型検査器のプロトタイプを実装
- ✅ ocaml-torch の例を移植して検査できることを確認
- ☐ Idris プログラムとの相互運用
- ☐ 多段階拡張

github.com/gfngfn/lw-staged-deptype