多段階計算による
コンパイル時テンソル形状検査

Takashi Suwa (諏訪 敬之)

gfngfn

@bd_gfngfn

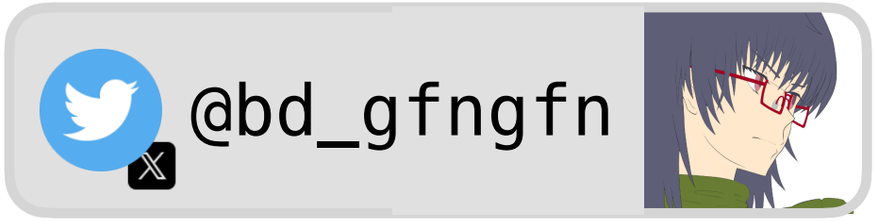
関数型まつり 2026　2026 年 7 月 11–12 日

於 中野セントラルパーク カンファレンス

はじめに

誰？

- サーバサイドエンジニア → 転職 D 進
- 主な制作物： **SATySFi**
  - https://github.com/gfngfn/SATySFi
  - 静的型つき関数型組版処理システム
  - IPA 未踏事業 2017 の 1 プロジェクト



内容：

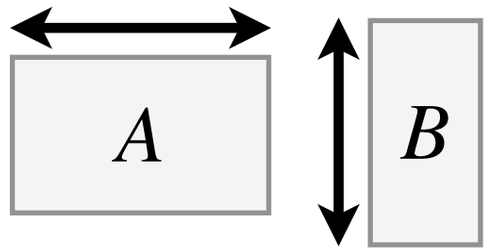
- 国際会議 **ECOOP 2026** にて発表した論文の宣伝です
  - 先週 6/29–7/3 にブリュッセル開催

English version is available here 👇

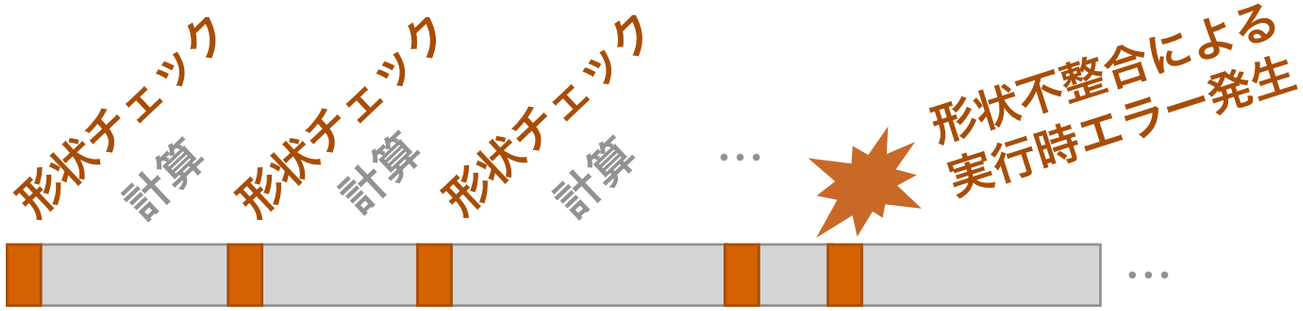
一般的な動機

- **テンソル**（ここでは**多次元配列**の意）
  - ベクトル，行列，…
  - 数理最適化アルゴリズムや DNN などで頻出

- テンソルを扱うプログラムを書く時は，
  テンソル形状の不整合が起きないことを**静的に**保証できるとよい
  - 例： 行列積 AB が意味をなすには
    (A の列数) = (B の行数) でなければならない



  - こうした保証がないと，**不整合は実行時エラーとなって現れ，数時間かかった計算が無駄になるかも**



よくある手法とその課題

以下のような型と定理証明機能によりテンソル形状を推論

- Vec n や Tensor $[l, m, n]$ といった形の**依存型**と手動証明
  - 例： Idris [Brady 2014]
- $\{\nu : \text{Tensor} \mid \text{len}(\nu\,.\,\text{shape}) = 3\}$ などの**篩型** + best-effort な自動証明
  - 例： Hybrid type checking [Flanagan 2006], GraTen [Hattori, Sato, & Kobayashi 2023]

よくある手法とその課題

以下のような型と定理証明機能によりテンソル形状を推論

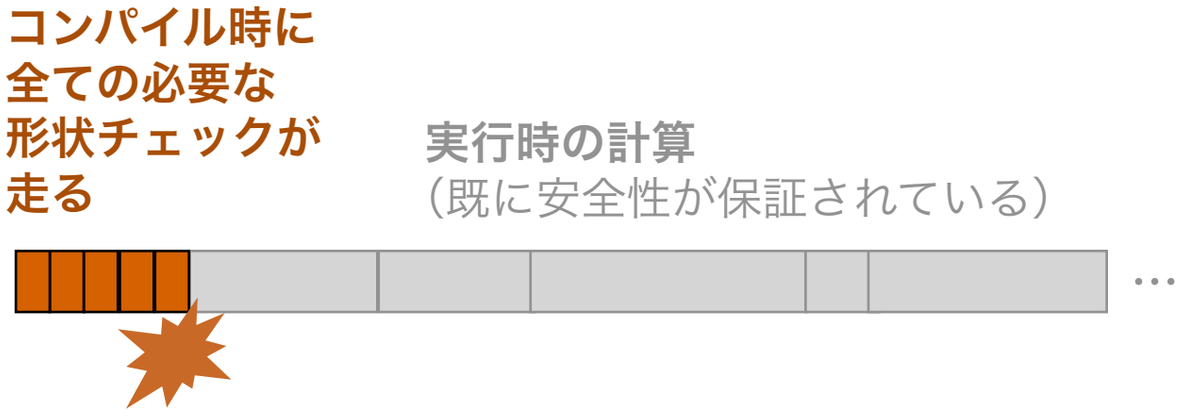
- Vec n や Tensor $[l, m, n]$ といった形の**依存型**と手動証明
  - 例： Idris [Brady 2014]
- $\{\nu : \text{Tensor} \mid \text{len}(\nu\,.\,\text{shape}) = 3\}$ などの**篩型** + best-effort な自動証明
  - 例： Hybrid type checking [Flanagan 2006], GraTen [Hattori, Sato, & Kobayashi 2023]

依然として残る課題： **継続的な**ソフトウェア開発との親和性

> 企業でのソフトウェア開発に於ける典型的状況：
>
> - ソフトウェアに求められる要件はユーザの好みや社会情勢により刻々と変化
> - 時は（文字通り）金なり！ 作業が手早く済むほど良い

- プログラムを修正するたびに証明の書き換えや SMT ソルバの機嫌取りが容易に発生するのは大きな負担

提案手法



定式化：

- **多段階計算** [Davies 1996] [Taha & Sheard 1997]
  と呼ばれる，**コンパイル時計算**と**実行時計算**を型安全に分離する体系をもとにして定義した“依存型の変種”を使う

検証方法：

- 型検査で挿入される**コンパイル時 assert 式**によって形状の整合性を保証
  - **不整合はコンパイル時計算の assertion failure 💥 として一瞬で発見される**
- 手動・自動問わず定理証明が不要
- assertion failure が起きずにコードが生成されたら，
  そのコードを実行しても不整合が起きないことが証明できている

目次

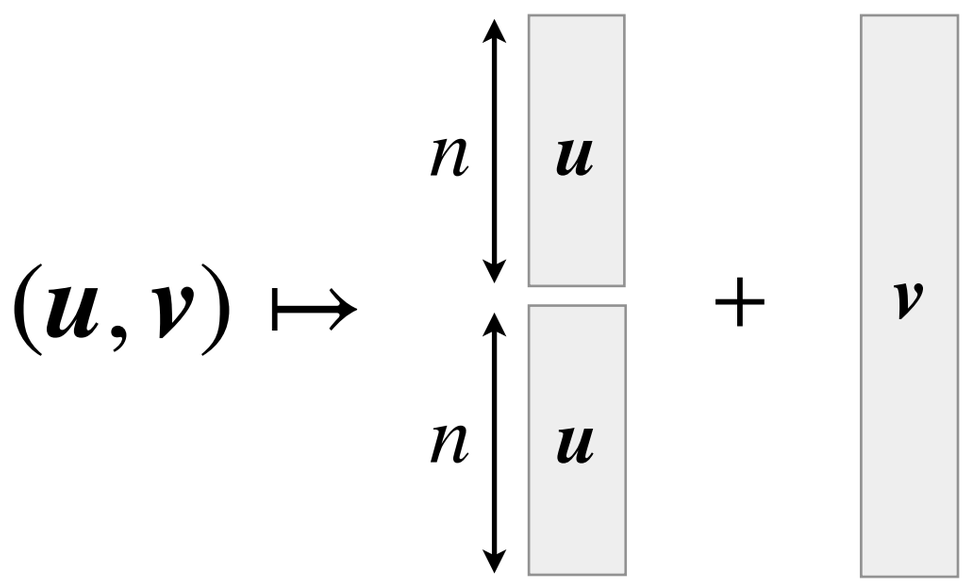
コア言語

- 形式化の詳細とさらなる改善
- 実装報告
- その他の議論

最小規模の例

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

```
repeat_and_add [| 10, 20, 30 |] [| 4, 5, 6, 7, 8, 9 |]
```

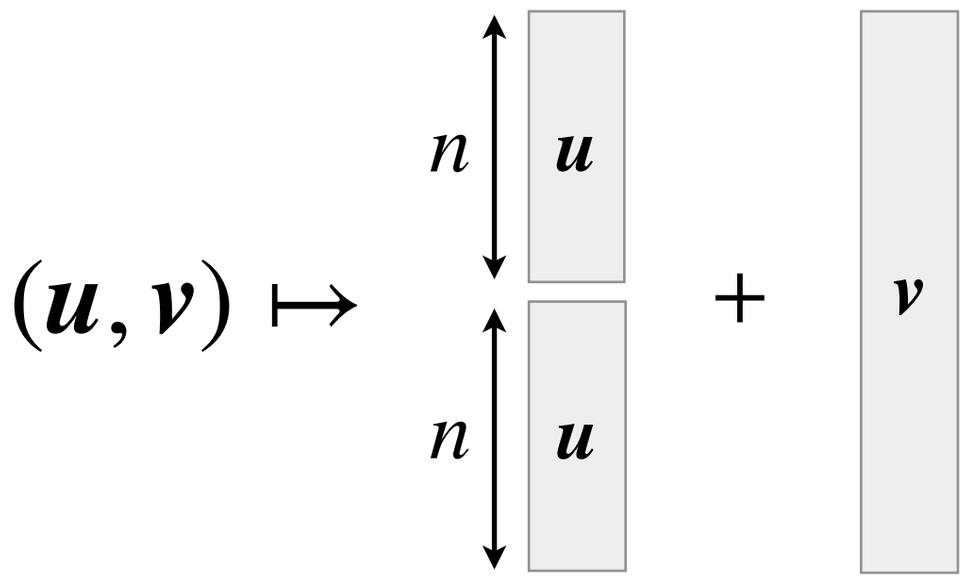


最小規模の例

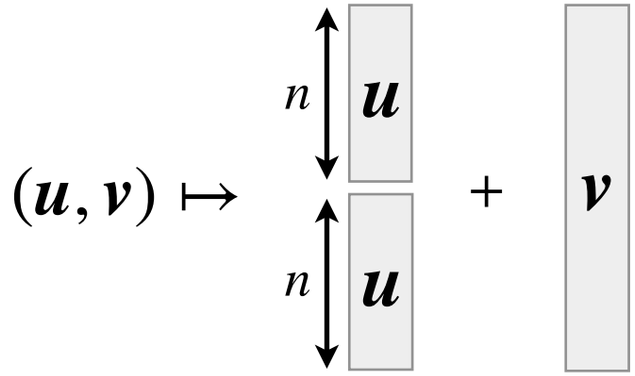
implicit パラメータ

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

```
repeat_and_add [| 10, 20, 30 |] [| 4, 5, 6, 7, 8, 9 |]
```



最小規模の例



こんな小さいプログラムでも型つけは非自明：

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

Vec (n + n)

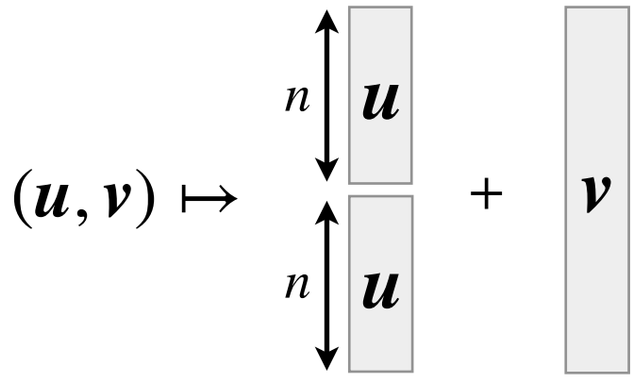
Vec (2 * n)

構文的に異なる型

```
vconcat : {k₁, k₂ : Nat} -> Vec k₁ -> Vec k₂ -> Vec (k₁ + k₂)

vadd : {k : Nat} -> Vec k -> Vec k -> Vec k
```

最小規模の例



2 つの型の等価性を言うために定理を証明したりするが,
これは前述の通り継続的な開発を妨げがち：

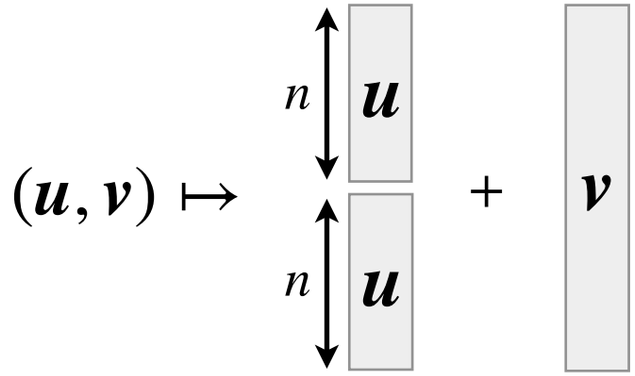
```
let repeat_and_add {n : Nat} (u : Vec n) (v : Vec (2 * n)) =
  vadd (vconcat u u) (rewrite mult_by_2_eq_add_self n in v)

theorem mult_by_2_eq_add_self (k : Nat) : 2 * k = k + k
proof
  …
end
```

…しかし，そもそもここで証明を要請すること自体が “過剰要求” かも

- 興味があるのは単に「少なくとも**実際に使われる形状の組み合わせに対して**正しく動くか」だったりする

最小規模の例

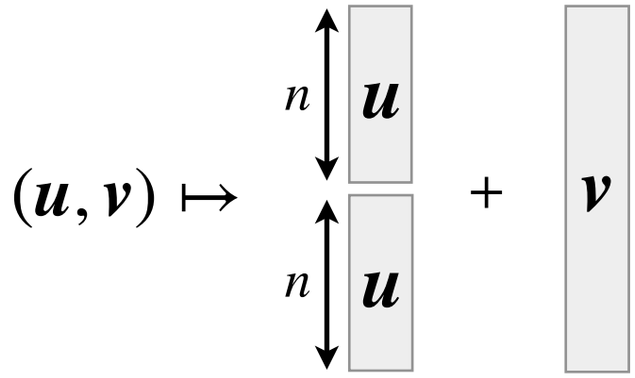


というわけで，**各インスタンスに対して**チェックする方針をとる：

```
let repeat_and_add {n : Nat} (u : Vec n) (v : Vec (2 * n)) =
  vadd (vconcat u u) (⦇Vec (2 * n) => Vec (n + n)⦈ v)
```

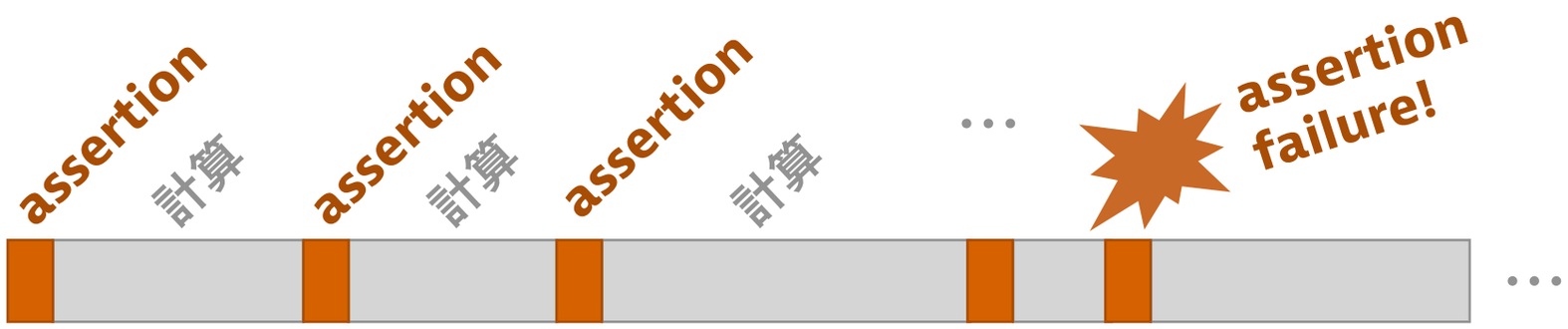
型： `Vec (2 * n) -> Vec (n + n)`
挙動： `assert (2 * n == n + n); (λx. x)`

最小規模の例




```
let repeat_and_add {n : Nat} (u : Vec n) (v : Vec (2 * n)) =
  vadd (vconcat u u) (⟪Vec (2 * n) => Vec (n + n)⟫ v)
```

- しかし，ここまでの変更では特に何も改善できていない
  - assertion はテンソルによる計算の合間に評価されるので，普通に数時間後に失敗しうる



- **必要な assertion だけ全部事前に計算することはできないか？**
  - …できます！（少なくとも典型的な DNN プログラムに関しては）
  - **多段階計算**を使って assertion が**コンパイル時に**評価されるようにしよう！

    [Davies 1996] [Taha & Sheard 1997]

目次

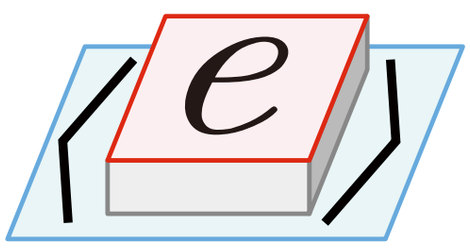
目次

多段階計算（ここでは 2 段階に限定）

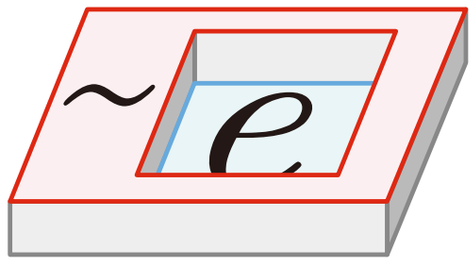
MetaML [Taha & Sheard 1997] 風の定式化：

bracket (quote)　**escape** (unquote)

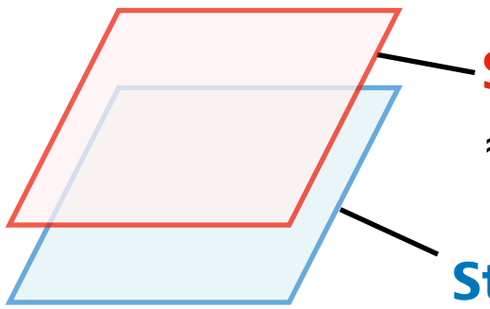
$$e ::= x \mid e\ e \mid \lambda x.\ e \mid \cdots \mid \langle e \rangle \mid {\sim}e$$



“**stage 1** で使われる
コード断片をつくる”



“**stage 0** で e がコード断片へと
評価され，穴を埋める”



- **stage 0** だけが通常の CBV で評価される
- コードの穴埋めは ~ と ⟨ ⟩ の相殺：

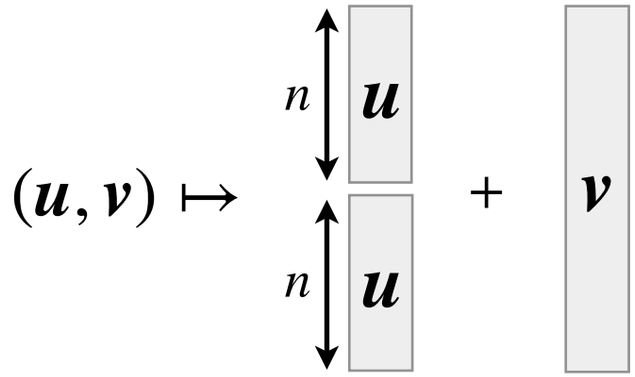
- プログラム全体を評価して穴のない $\langle e \rangle$ の形に
なったら，その e が通常のプログラムとして使われる

$$\tau ::= \tau \to \tau \mid \cdots \mid \langle \tau \rangle$$

コード型

目次

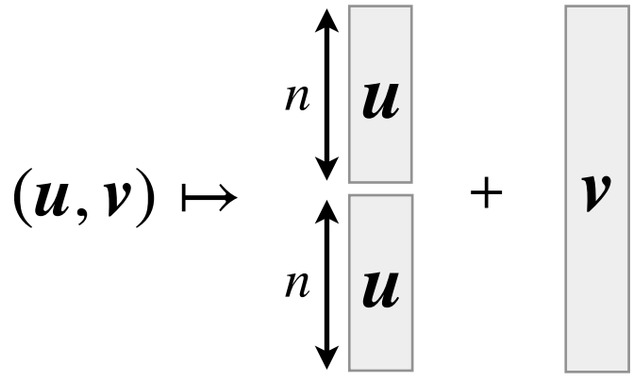
Assertion がコンパイル時に評価されるように修正



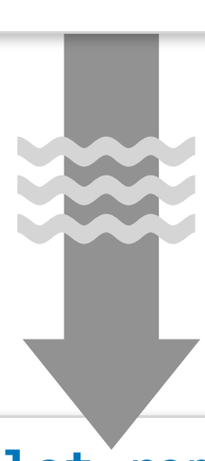
まず仮引数を λ 抽象へと脱糖：

```
let repeat_and_add = λ{n : Nat}. λ(u : Vec n). λ(v : Vec (2 * n)).
  vadd
    (vconcat u u)
    ((|Vec (2 * n) => Vec (n + n)|) v)
```

Assertion がコンパイル時に評価されるように修正



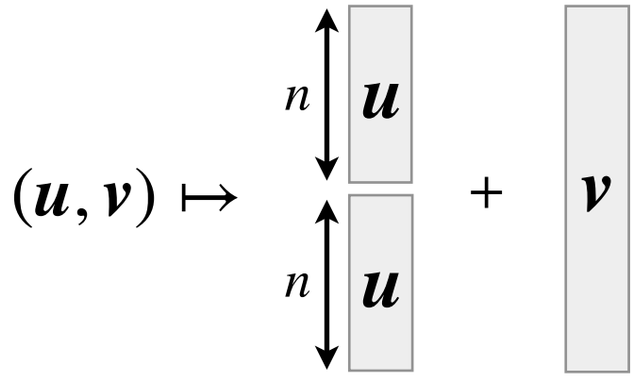

```
let repeat_and_add = λ{n : Nat}. λ(u : Vec n). λ(v : Vec (2 * n)).
  vadd
    (vconcat u u)
    ((|Vec (2 * n) => Vec (n + n)|) v)
```



最終的に以下が得られる（以降で順を追って説明）

```
let repeat_and_add = λ{n : Nat}. ⟨λ(u : Vec %n). λ(v : Vec %(2 * n)).
  ~(gen_vadd {n + n})
    (~(gen_vconcat {n} {n}) u u)
    ~((|⟨Vec %(2 * n)⟩ => ⟨Vec %(n + n)⟩|) ⟨v⟩)
⟩
```

Assertion がコンパイル時に評価されるように修正



assertion は **stage 0** にあってほしく,
一方で実際のテンソルの計算は **stage 1** にあってほしい：

```
let repeat_and_add = λ{n : Nat}. λ(u : Vec n). λ(v : Vec (2 * n)).
  vadd
    (vconcat u u)
    ((|Vec (2 * n) => Vec (n + n)|) v)
```

Assertion がコンパイル時に評価されるように修正

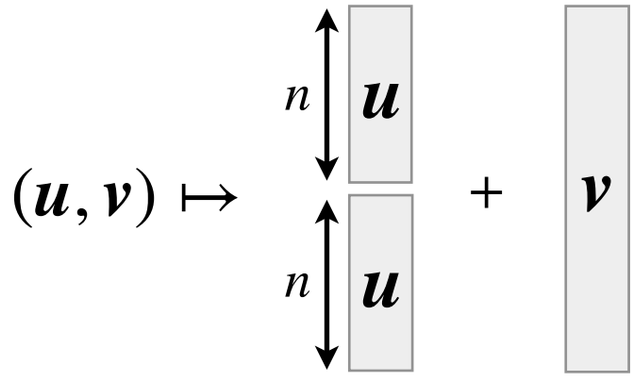
$$(u, v) \mapsto \begin{matrix} n \updownarrow u \\ n \updownarrow u \end{matrix} + v$$

所望のステージになるように，assertion 周りに ~ と ⟨⟩ を挿入

```
let repeat_and_add = λ{n : Nat}. λ(u : Vec n). λ(v : Vec (2 * n)).
  vadd
    (vconcat u u)
    ~((⟨Vec (2 * n)⟩ => ⟨Vec (n + n)⟩) ⟨v⟩)
```

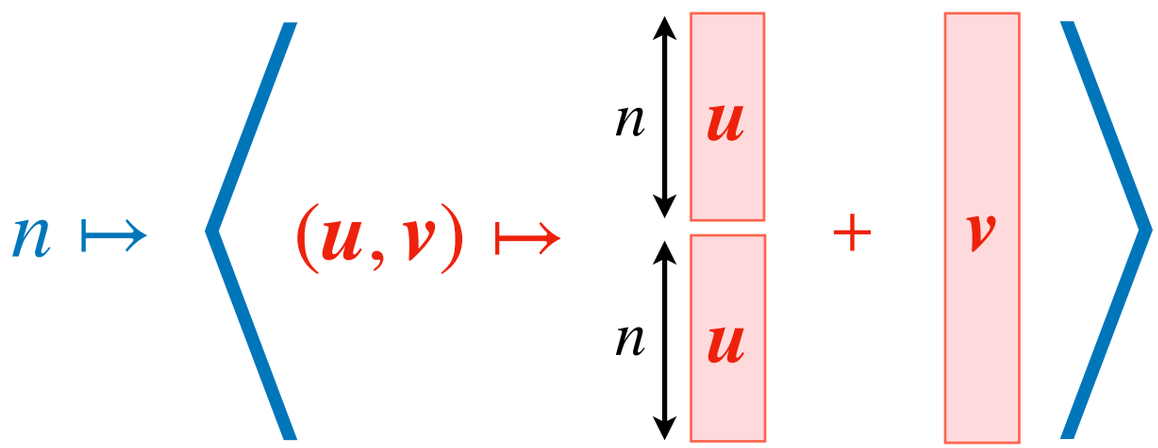
- assertion はコード型間のキャストになる
- 残る重要な点：
  - `2 * n` と `n + n` は**コンパイル時**に計算して比較したいので，
    変数 `n` は **stage 0** にあるべき

Assertion がコンパイル時に評価されるように修正

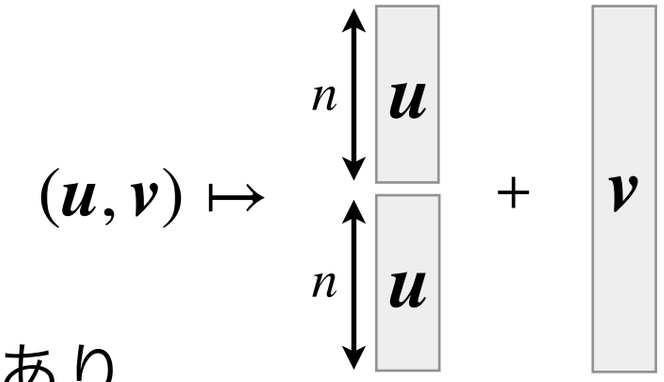



```
let repeat_and_add = λ{n : Nat}. ⟨λ(u : Vec %n). λ(v : Vec %(2 * n)).
  vadd
    (vconcat u u)
    ~((⦇⟨Vec %(2 * n)⟩ => ⟨Vec %(n + n)⟩⦈ ⟨v⟩)
⟩
```

- `Vec %n` は **stage 1** の型である一方，引数 `n` は **stage 0**
  - `%` は単なる記号で，**cross-stage persistence** [Taha+ 2000] とのアナロジーのために付加
- `repeat_and_add` は今やマクロのような姿に：



Assertion がコンパイル時に評価されるように修正



実際は vadd と vconcat も長さの引数を取る必要があり，
そのために gen_vadd と gen_vconcat という関数で扱う：

```
let repeat_and_add = λ{n : Nat}. ⟨λ(u : Vec %n). λ(v : Vec %(2 * n)).
  ~(gen_vadd {n + n})
    (~(gen_vconcat {n} {n}) u u)
    ~((|⟨Vec %(2 * n)⟩ => ⟨Vec %(n + n)⟩|) ⟨v⟩)
⟩
```

- gen_vadd : $\{k : \mathrm{Nat}\} \to \langle \mathrm{Vec}\ \%k \to \mathrm{Vec}\ \%k \to \mathrm{Vec}\ \%k \rangle$
  - Returns $\mathrm{vadd}_k : \mathrm{Vec}\ k \to \mathrm{Vec}\ k \to \mathrm{Vec}\ k$ (if $k \geq 0$)
- gen_vconcat : $\{k : \mathrm{Nat}\} \to \{l : \mathrm{Nat}\} \to \langle \mathrm{Vec}\ \%k \to \mathrm{Vec}\ \%l \to \mathrm{Vec}\ \%(k + l) \rangle$
  - Produces $\mathrm{vconcat}_{k,l} : \mathrm{Vec}\ k \to \mathrm{Vec}\ l \to \mathrm{Vec}\ m$ (where $m := k + l$)

評価例

```
let repeat_and_add = λ{n : Nat}. ⟨λ(u : Vec %n). λ(v : Vec %(2 * n)).
  ~(gen_vadd {n + n})
    (~(gen_vconcat {n} {n}) u u)
    ~(⌊⟨Vec %(2 * n)⟩ => ⟨Vec %(n + n)⟩⌋ ⟨v⟩)
⟩
```

```
⟨~(repeat_and_add {3}) [| 10, 20, 30 |] [| 4, 5, 6, 7, 8, 9 |]⟩
```

コンパイル時計算（大きなプログラムでも典型的には一瞬）

この例では `2 * 3 == 3 + 3` が評価され無事通る

```
⟨(λ(u : Vec %3). λ(v : Vec %6). vadd6 (vconcat3,3 u1 u2) v)
    [| 10, 20, 30 |]
    [| 4, 5, 6, 7, 8, 9 |]
⟩
```

実行時計算（ここは実際的なプログラムだと数時間規模）

ここで形状不整合で失敗することは決してない！

```
[| 14, 25, 36, 17, 28, 39 |]
```

目次

型検査時の assertion 挿入

```
let repeat_and_add = λ{n : Nat}. ⟨λ(u : Vec %n). λ(v : Vec %(2 * n)).
  ~(gen_vadd {n + n}) (~(gen_vconcat {n} {n}) u u) v
⟩
```

2 つの型が compatible だが構文的に一致しない場合に挿入

```
let repeat_and_add = λ{n : Nat}. ⟨λ(u : Vec %n). λ(v : Vec %(2 * n)).
  ~(gen_vadd {n + n})
    (~(gen_vconcat {n} {n}) u u)
    ~(⦇⟨Vec %(2 * n)⟩ => ⟨Vec %(n + n)⟩⦈ ⟨v⟩)
⟩
```

- 簡略化すると以下のような elaboration をやる：

$$\frac{\Gamma \vdash M_1 : (\mathsf{Vec}\ \%\, N_{11} \to T) \rightsquigarrow E_1 \qquad \Gamma \vdash M_2 : \mathsf{Vec}\ \%\, N_2 \rightsquigarrow E_2}{\Gamma \vdash M_1\ M_2 : T \rightsquigarrow (E_1\ \sim(⦇\langle \mathsf{Vec}\ \%\, N_2 \rangle \Rightarrow \langle \mathsf{Vec}\ \%\, N_{11} \rangle⦈\ \langle E_2 \rangle))}$$

- Cf. よくある依存型システムでの型つけ規則：

$$\frac{\Gamma \vdash M_1 : \mathsf{Vec}\ N_{11} \to T \qquad \Gamma \vdash M_2 : \mathsf{Vec}\ N_2 \qquad \vDash \forall \Gamma\ .\ N_2 = N_{11}}{\Gamma \vdash M_1\ M_2 : T}$$

定式化の概要

$$\text{Vec}\ \%M^{(0)} := \text{Tensor}\ \%[M^{(0)}]$$

型式：

$$B ::= \text{Bool} \mid \text{Int} \mid \text{NatList} \mid \cdots$$

$$T^{(1)} ::= B \mid T^{(1)} \to T^{(1)} \mid \text{Tensor}\ \%M^{(0)}$$

$$T^{(0)} ::= \{x : B \mid M^{(0)}\} \mid \langle T^{(1)} \rangle \mid (x : T^{(0)}) \to T^{(0)} \mid \{x : T^{(0)}\} \to T^{(0)} \mid \cdots$$

引数の式は **stage 0**

挿入される assertion:

$$M^{(0)} ::= \cdots \mid (\!| \langle T^{(1)} \rangle \Rightarrow \langle T^{(1)} \rangle |\!) \mid (\!| \Rightarrow \{x : B \mid N^{(0)}\} |\!)$$

形状整合性チェック用

定式化の概要

$$\text{Vec}\ \%M^{(0)} := \text{Tensor}\ \%[M^{(0)}]$$

型式：

$$B ::= \text{Bool} \mid \text{Int} \mid \text{NatList} \mid \cdots$$

$$T^{(1)} ::= B \mid T^{(1)} \to T^{(1)} \mid \text{Tensor}\ \%M^{(0)}$$

$$T^{(0)} ::= \{x : B \mid M^{(0)}\} \mid \langle T^{(1)} \rangle \mid (x : T^{(0)}) \to T^{(0)} \mid \{x : T^{(0)}\} \to T^{(0)} \mid \cdots$$

引数の式は **stage 0**

stage 0 にだけ篩型があり，
broadcasting などを表すのに使える

$$\text{Tensor.gen\_add} : \{s_1 : \text{NatList}\} \to \{s_2 : \{\nu : \text{NatList} \mid \text{broadcastable}\ s_1\ \nu\}\} \to \langle \text{Tensor}\ \%s_1 \to \text{Tensor}\ \%s_2 \to \text{Tensor}\ \%(\text{broadcast}\ s_1\ s_2) \rangle$$

挿入される assertion:

$$M^{(0)} ::= \cdots \mid (\!| \langle T^{(1)} \rangle \Rightarrow \langle T^{(1)} \rangle |\!) \mid (\!| \Rightarrow \{x : B \mid N^{(0)}\} |\!)$$

形状整合性チェック用

篩型の述語に関するチェック用

Metatheory

Theorem (***Soundness of Insertion***). $\Gamma \vdash^n M^{(n)} : T^{(n)} \rightsquigarrow N^{(n)}$ implies $\Gamma \Vdash^n N^{(n)} : T^{(n)}$.

Theorem. (***Preservation***). $\Gamma \Vdash^n N^{(n)} : T^{(n)}$ and $N^{(n)} \longrightarrow^n N'^{(n)}$ imply $\Gamma \Vdash^n N'^{(n)} : T^{(n)}$.

Theorem. (***Progress***). If $\vdash^1 \Gamma$ and $\Gamma \Vdash^n N^{(n)} : T^{(n)}$, then one of the following holds: (1) $N^{(n)}$ is a value, (2) $N^{(n)} \longrightarrow^n \mathsf{err}$, or (3) $N^{(n)} \longrightarrow^n N'^{(n)}$ for some $N'^{(n)}$.

Theorem. (***Generated code is "simply-typed with infinite num. of base types"***). If $\vdash_{\mathrm{wf}} \Gamma$, $\Gamma \Vdash^1 v^{(1)} : T^{(1)}$, and $\Gamma \equiv^1 \gamma$, then there exists $\tau^{(1)}$ such that $\gamma \vdash v^{(1)} : \tau^{(1)}$ and $T^{(1)} \equiv^1 \tau^{(1)}$.

$$\gamma ::= \cdot \mid \gamma, x : \tau^{(1)} \qquad \tau^{(1)} ::= B \mid \mathsf{Tensor}\ \%[n, \ldots, n] \mid \tau^{(1)} \to \tau^{(1)}$$

型の同値関係 $\equiv^n$ をどう定義するかはかなり非自明！

- 組込み関数の意味論と篩型とを整合させるために工夫を要する
- 単なる βη 同値では緩すぎる
- 結局 **common subexpression reduction** (**CSR**) [Greenberg 2013] [Sekiyama+ 2017] という二項関係が使えた

さらなる改善

- まだまだ冗長な記述がたくさんある
- 実は，最初に掲げた以下のプログラムの書き方を表層言語にできる！

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

  - 表層言語には **Horsea** と命名

- 以下により実現：
  1. ステージの推論
     - **束縛時解析** [Jones+ 1993] [Davies 1996] という古典的手法で可能
  2. 明示されていない implicit 引数を **best-effort で**復元
     - “**Let Arguments Go First**” [Xie & Oliveira 2018] と呼ばれる **bidirectional type-checking** [Dunfield 2013] の変種を通じて復元

```
⟨~(gen_vconcat {n} {n}) u u)⟩
```

```
⟨~(repeat_and_add {3}) [| 10, 20, 30 |] [| 4, 5, 6, 7, 8, 9 |]⟩
```

目次

実装報告

- 型検査器の Haskell 製プロトタイプ実装：
  - https://github.com/gfngfn/Horsea
- **ocaml-torch** のリポジトリにある 10 個のプログラム例を移植し，その形状整合性が検証できることを確認
- 意図的に簡単なものになっている推論アルゴリズムにもかかわらず，90% ほどの implicit 引数が復元できることを確認

Program	#lines	inferred/all	#annots
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

目次

提案手法のねらい

1. 形状不整合のエラーが実行時に決して起きないことを
   言語水準の metatheory で保証
2. **broadcasting** などの複雑な変換も扱えるようにする
3. 継続的なソフトウェア開発ワークフローとの親和性
   (A) “偽陽性” の型エラーが出過ぎない
   (B) 検証にかかる時間が短い，または予測しやすい

- 知る限り，上記全部を満たす既存手法はなかった
- 多段階計算ベースの提案手法は或る種の解決になっている
  - 数学的保証あり
  - 篩型によって broadcasting などの変換も扱える
  - 値による比較なので，原理的に “偽陽性” のエラーがない
  - 自動証明に依存しないので，ユーザの手が及ばない部分での所要時間の心配なし

提案手法の暗黙的前提

「典型的なプログラムではテンソルの形状はそれほど動的ではなく,
入力の形状が決まれば出力の形状も定まるような演算のみを用いて書ける」という前提に依存している

- 例： A と B の形状がそれぞれ事前にわかっていれば,
  行列積 AB の形状は実際に計算せずとも事前にわかる
- 例： 一方で, 行列を受け取って `List.filter` の要領で
  望ましい列だけ取り出すような演算は,
  実際に計算するまで出力の形状を特定できない

提案手法の限界

- 評価によって真偽を判定できない性質の保証には使えない
  - 例： 本質的に全称量化されている命題
  - したがって，ライブラリの検証には不向き
    - property-based testing のような使い方はできるかも
- 実行時に初めて形状がわかるようなテンソルも扱えはするが，
  不整合が生じる可能性を排除することはできない
  - これは単に**実行時にコード生成**して即座にそのコードを実行するだけ
  - しかし依然として，**assertion failure はコード生成時に生じるのであって実際のテンソルを使った計算のさなかに起きるわけではない**ので，大幅にマシ

主な関連研究との比較

- 操作的には，提案手法は **LMS-Verify** [Amin & Rompf 2017] とよく似る
  - 差分：
    - **言語水準での**安全性保証
    - 型つけ主導での assertion の自動挿入
  - 型でテンソル形状を持ち回っているのがこれらの利点に効いている

- **GraTen** [Hattori, Sato, & Kobayashi 2024]
  - 篩型を用いてテンソル形状を best-effort に推論
  - broadcasting のサポートや assertion 挿入に関して大いに参考にしました

- **Idris 2** [Brady 2021]
  - **QTT** [Atkey 2018] の multiplicity に基づく，ステージのような区分がある
  - interoperability を研究してみるのも面白いかも

まとめ

- **コンパイル時に**テンソル形状検査を行なう定式化を**多段階計算**に基づいて与え，その型安全性を証明した
  - コード生成に成功すれば，実行時には決して形状不整合が起きない
  - 最大の提案動機： 継続的なソフトウェア開発との親和性
- その他提案：
  - ステージなしの表層言語 **Horsea**
  - implicit 引数を best-effort で復元する，bidirectional type-checking に基づくアルゴリズム
- Haskell 製プロトタイプ実装：
  - https://github.com/gfngfn/Horsea