Compile-Time Tensor Shape Checking via Staged Shape-Dependent Types

Takashi Suwa*† **Atsushi Igarashi***

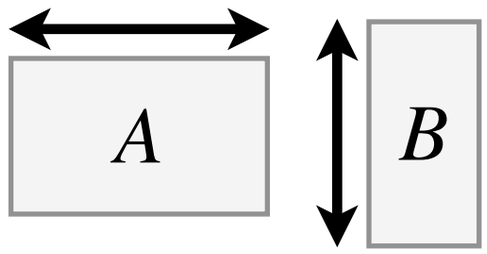
*: Kyoto University †: Imiron Co., Ltd.

ECOOP 2026 June 29 – July 3, 2026, Brussels

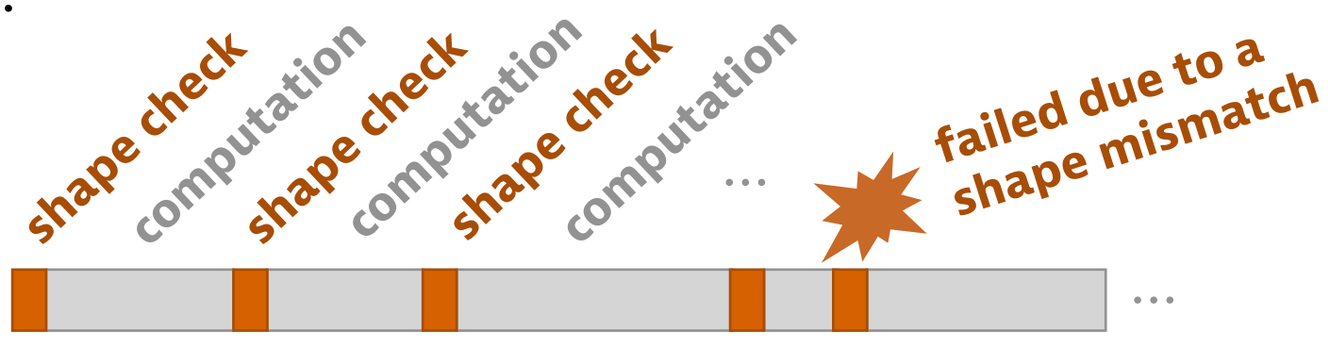
General Motivation

- ***Tensors***, also called as ***multi-dimensional arrays***
  - I.e., vectors, matrices, …
  - Used for mathematical optimization, deep neural networks (DNNs), etc.

- When writing tensor-handling programs,
  it is desirable to ***statically rule out tensor shape mismatches***
  - E.g., for matrix multiplication AB,
    we must have $\#\mathrm{cols}(A) = \#\mathrm{rows}(B)$



  - ***Otherwise, mismatches will be detected only as runtime errors***,
    possibly a few hours later!



Major Approach & Further Issues

With the aid of theorem proving, statically reasons shapes by using:

- ***dependent types*** like Vec n or Tensor $[l, m, n]$
  - E.g., `Vect` in *Idris* [Brady 2014]
- ***refinement types*** like $\{\nu : \text{Tensor} \mid \text{len}(\nu\,.\,\text{shape}) = 3\}$
  and automated, best-effort proving
  - E.g., *Hybrid type checking* [Flanagan 2006], *GrαTen* [Hattori, Sato, & Kobayashi 2023]

Major Approach & Further Issues

With the aid of theorem proving, statically reasons shapes by using:

- ***dependent types*** like Vec n or Tensor $[l, m, n]$
  - E.g., `Vect` in *Idris* [Brady 2014]
- ***refinement types*** like $\{\nu : \text{Tensor} \mid \text{len}(\nu\,.\,\text{shape}) = 3\}$
  and automated, best-effort proving
  - E.g., *Hybrid type checking* [Flanagan 2006], *GrαTen* [Hattori, Sato, & Kobayashi 2023]

Possible remaining gap: affinity with ***continuous software dev.***

Typical circumstances in industry:

- Requirements on software easily change in an ***unexpected*** way, depending on users' preference or social environment
- Time is (literally) money; sooner is better

- Modification might be frequently bothered by updating proofs or taming uncertainty of proof automation

Our Approach

Formalization:

- Use shape-dependent types based on ***staging***, [Davies 1996] [Taha & Sheard 1997] which safely separates ***compile-time*** computation from ***runtime***

Method:

- Verifies shape consistency ***through compile-time assertions*** that are inserted by type-checking
  - Mismatches are ***virtually statically*** detected as compile-time failures
- No manual/automated proving is necessary
- ***Runtime safety is still guaranteed*** once code generation succeeds

(Let us discuss related work later!)

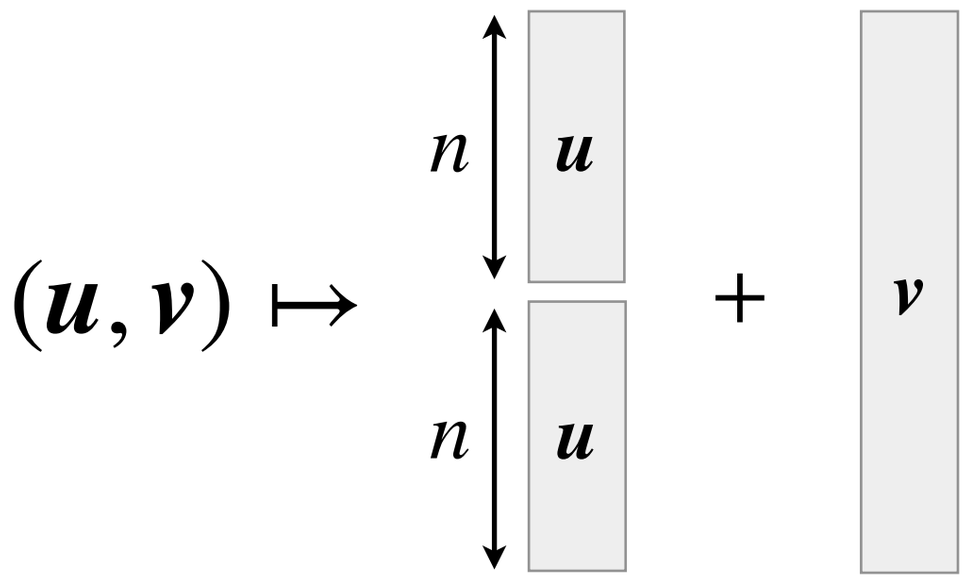
Outline

- **Our core language**
- Formalization and further improvement
- Implementation report
- Discussions

Minimal Example

```
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 |]
```

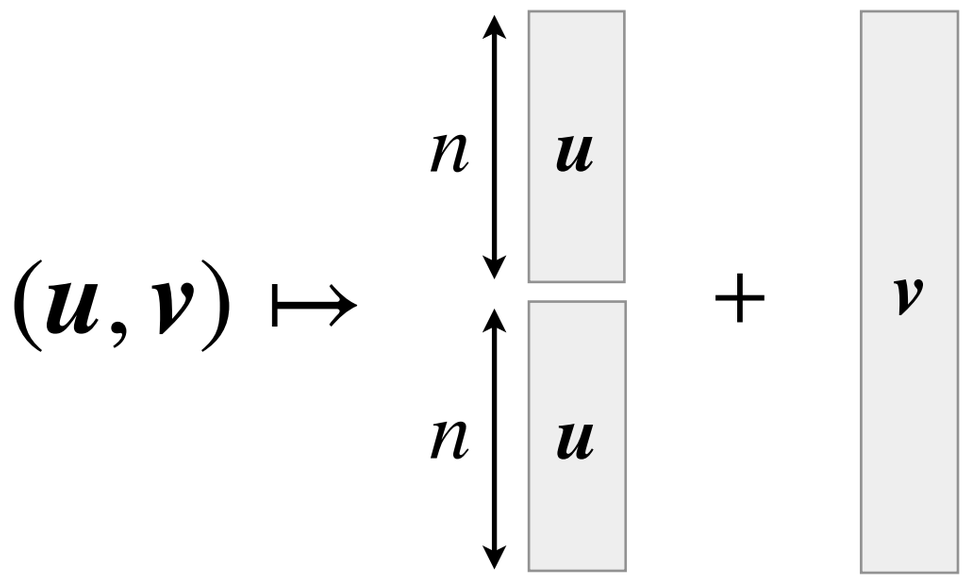


Minimal Example

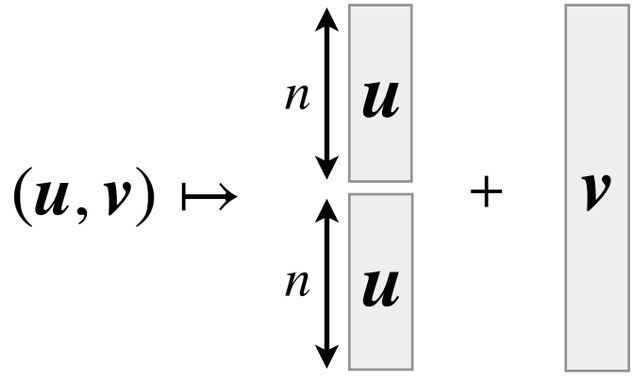
implicit parameter

```
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 |]
```



Minimal Example



Type-checking is not trivial:

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

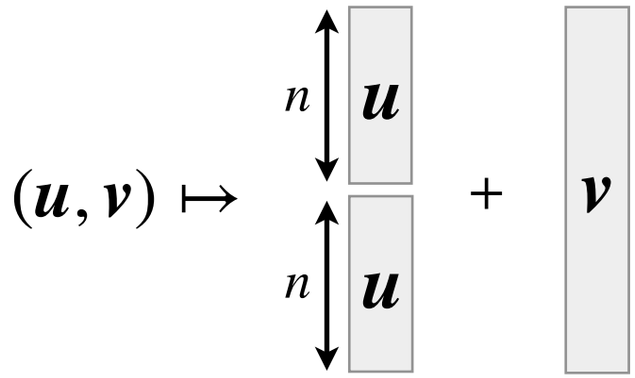
`Vec (n + n)` `Vec (2 * n)`

Not syntactically identical

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

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

Minimal Example



Often have to write proofs to equate two types,
which might easily hamper continuous development:

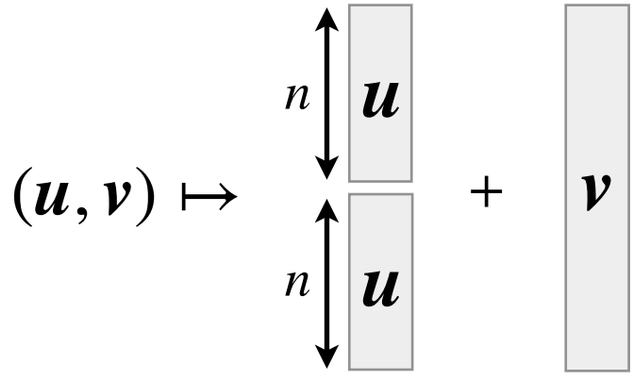
```
let repeat_and_add {n : Nat} (u : Vec n) (v : Vec (2 * n)) =
  vadd (vconcat u u) (rewrite mult_by_2_eq_add_self n in v)

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

…But, often, such proofs are “overkill”

- Interest might be just whether
  each function works ***at least for actually used instances***

Minimal Example

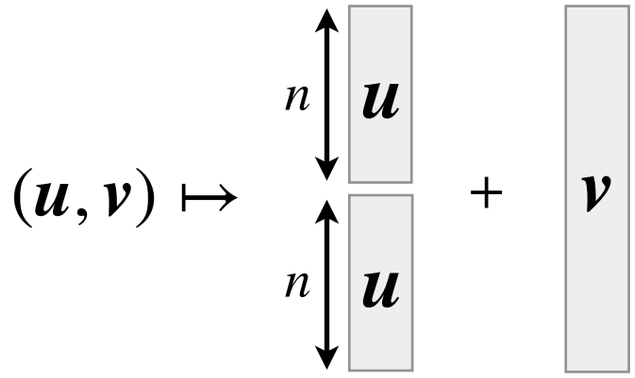


Thus, consider doing assertion on shapes ***for each instance***:

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

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

Minimal Example




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

- However, assertion failures will happen at runtime, possibly after few hours!

- ***Can we collect all the assertion phases and execute them beforehand?***
  - …Yes, at least for typical DNN programs
  - Let's make assertions evaluated at ***compile time***, using ***staged computation***

[Davies 1996] [Taha & Sheard 1997]

Outline

- **Our core language**
  - **Introduction**
    - Preliminaries: staged computation
    - Making assertions evaluated at compile time
  - Formalization and further improvement
  - Implementation report
  - Discussions

Outline

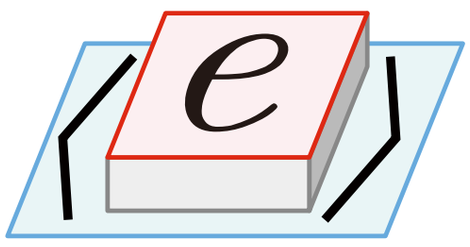
- **Our core language**
  - Introduction
  - **Preliminaries: staged computation**
  - Making assertions evaluated at compile time
- Formalization and further improvement
- Implementation report
- Discussions

Staged Computation (Two Stage)

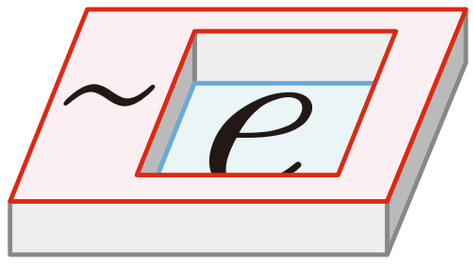
A minimal language similar to ***MetaML*** [Taha & Sheard 1997]:

bracket (quote)　　***escape*** (unquote)

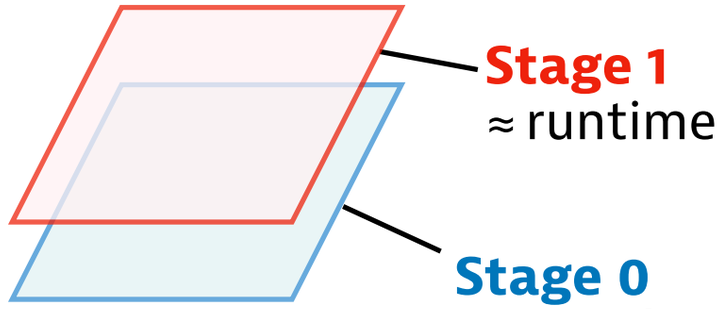
$$e ::= x \mid e\ e \mid \lambda x.\ e \mid \cdots \mid \langle e \rangle \mid {\sim}e$$



"Forms a code fragment that works at **stage 1**"



"Splices a code fragment produced by e at **stage 0**"



- Only **stage 0** is evaluated by ordinary CBV
- Splicing happens by canceling ~ and ⟨ ⟩:

- The whole program reaches $\langle e \rangle$ with no holes, and then e is used as a runtime program

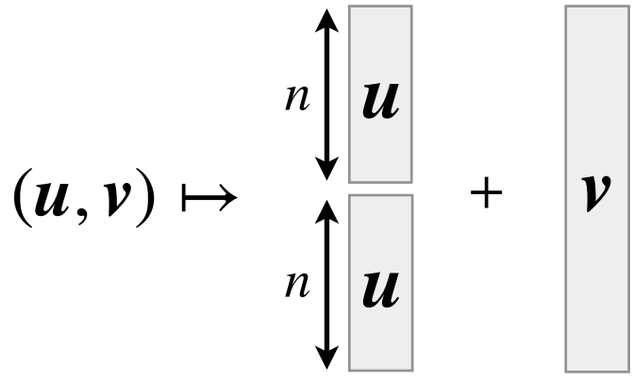
$$\tau ::= \tau \to \tau \mid \cdots \mid \langle \tau \rangle$$

code type

Outline

- **Our core language**
  - Introduction
  - Preliminaries: staged computation
  - **Making assertions evaluated at compile time**
- Formalization and further improvement
- Implementation report
- Discussions

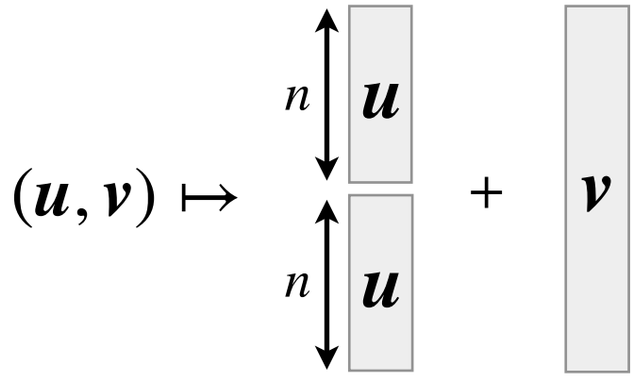
Making Assertions Evaluated at Compile Time



Desugars parameters to λ-abstractions:

```
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)
```

Making Assertions Evaluated at Compile Time

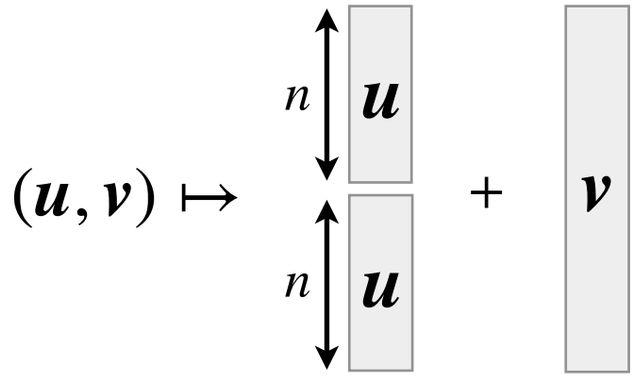



```
let repeat_and_add = λ{n : Nat}. λ(u : Vec n). λ(v : Vec (2 * n)).
  vadd
    (vconcat u u)
    ((|Vec (2 * n) => Vec (n + n)|) v)
```

We will see the following is finally obtained:

```
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⟩)
⟩
```

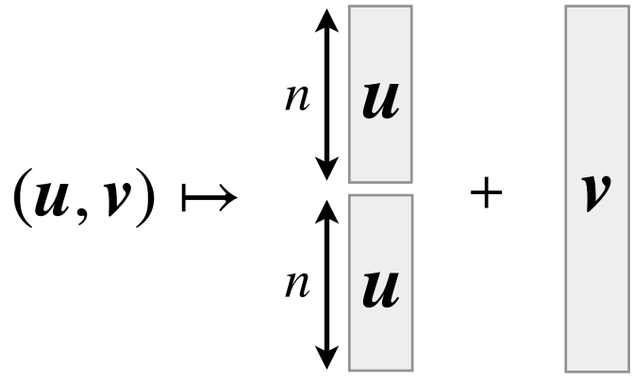
Making Assertions Evaluated at Compile Time



We want assertions to be at **stage 0**,
while tensor-handling computation should be at **stage 1**:

```
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)
```

Making Assertions Evaluated at Compile Time

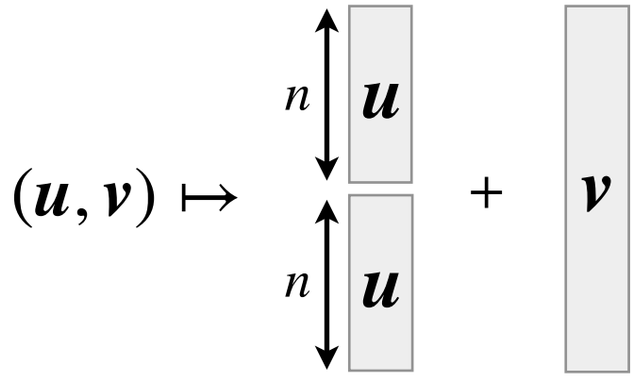


Inserted ~ and ⟨ ⟩ around the assertion:

```
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⟩)
```

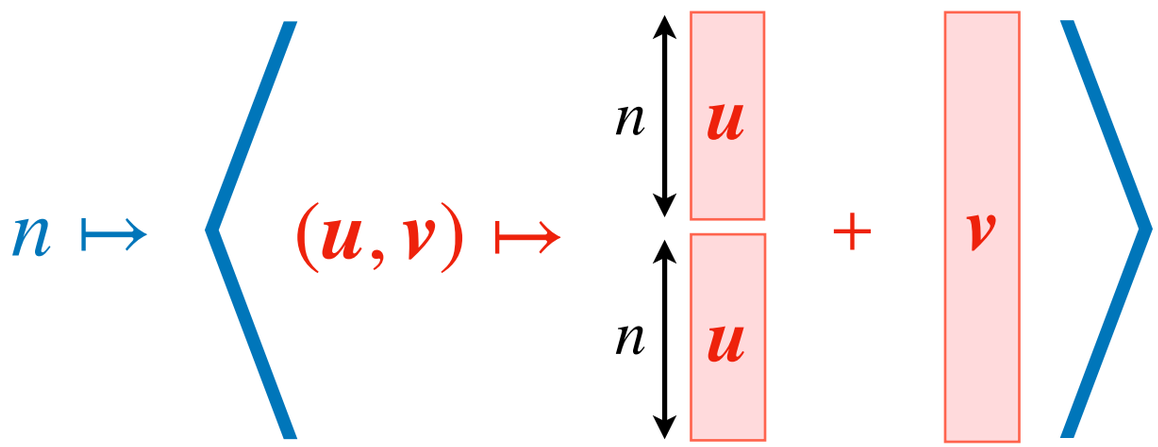
- Now, casts are between two code types
- Remaining essential point:
  - `2 * n` and `n + n` must be computable at **compile time**, so `n` must live in **stage 0**

Making Assertions Evaluated at Compile Time

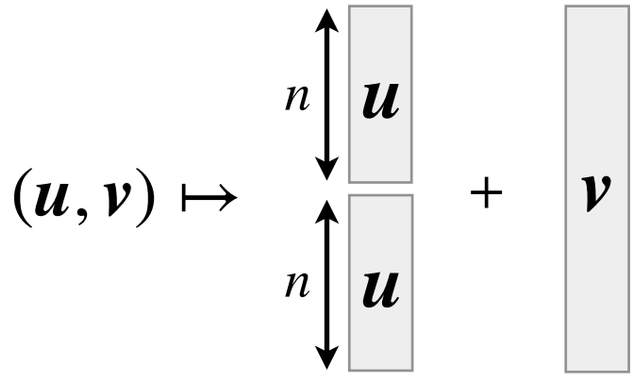



```
let repeat_and_add = λ{n : Nat}. ⟨λ(u : Vec %n). λ(v : Vec %(2 * n)).
  vadd
    (vconcat u u)
    ~((⌈⟨Vec %(2 * n)⟩ => ⟨Vec %(n + n)⟩⌋ ⟨v⟩)
⟩
```

- While `Vec %n` is a **stage-1** type, `n` is at **stage 0**
  - `%` is just a symbol for an analogy with ***cross-stage persistence*** [Taha+ 2000]
- `repeat_and_add` now works like a macro:



Making Assertions Evaluated at Compile Time



vadd and vconcat also need to take lengths,
so they should be gen_vadd and gen_vconcat:

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

Example Evaluation

```
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 |]⟩
```

Compile-time computation (typically instant)

…In this case, the assertion `2 * 3 == 3 + 3` passes

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

Runtime (likely to be heavy for large programs)

…***No possiblity of failure here!***

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

Outline

- Our core language
  - Introduction
  - Preliminaries: staged computation
  - Making assertions evaluated at compile time
  - **Formalization and further improvement**
- Implementation report
- Discussions

Assertion Insertion through Typing

```
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
⟩
```

Insert assertions where two types are compatible but different

```
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⟩)
⟩
```

- Our elaboration is like:

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

- Cf. A conventional rule (simplified):

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

Basics of Our Formalization

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

Type expressions:

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

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

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

Argument expressions must be at **stage 0**

Assertions inserted by elaboration:

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

For shape consistency

Basics of Our Formalization

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

Type expressions:

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

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

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

Argument expressions must be at **stage 0**

Only **stage 0** has refinement types for conversions such as ***broadcasting***

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

Assertions inserted by elaboration:

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

For shape consistency

For refinement predicates

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 $T^{(1)} \equiv^1 \tau^{(1)}$.

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

How to define the equivalence $\equiv^n$ was quite non-trivial!

- due to the interaction of primitives and refinement types
- We used ***common subexpression reduction*** (***CSR***) [Greenberg 2013] [Sekiyama+ 2017]

Further Improvements

- There still remains much redundancy
- Actually, ***we can provide the following as a surface language Horsea***

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

- To this end, we achieved:

1. Inference of stages
   - Can be done (at least) by ***binding-time analysis*** [Jones+ 1993] [Davies 1996]
2. Reconstruction of implicit arguments ***in a best-effort manner***, guided by types by using "***Let Arguments Go First***," [Xie & Oliveira 2018] a variant of ***bidirectional type-checking*** [Dunfield 2013]

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

```
⟨~(gen_vconcat {n} {n}) u u)⟩
```

Outline

- Our core language
  - Introduction
  - Preliminaries: staged computation
  - Making assertions evaluated at compile time
- Formalization and further improvement
- **Implementation report**
- Discussions

Implementation Report

- Wrote a prototype type-checker in Haskell
  - https://github.com/gfngfn/Horsea
- Ported 10 example programs offered by ***ocaml-torch*** into Horsea, and confirmed that their consistency can be verified
- Succeeded in reconstructing ~90% of implicit arguments
  - …although our inference algorithm is (intentionally) quite simple!

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

Outline

- Our core language
  - Introduction
  - Preliminaries: staged computation
  - Making assertions evaluated at compile time
- Formalization and further improvement
- Implementation report
- **Discussions**

Precise Description of Our Goals

1. Achieve a ***language-level, mathematical guarantee*** that mismatches never happen during runtime
2. Allow complex tensor conversions like ***broadcasting***
3. Affinity with ***continuous*** software development
   (A) Does not cause “false positive” type errors too easily
   (B) Has reasonable (or at least predictable) time consumption

- As far as we know, existing methods lack some of the above
- Our staging-based formalization gives some sort of solution!

Our Method

- Achieves all of our goals:
  - Fulfills the mathematical guarantee (which was tough to prove!)
  - Accommodates complex conversions by refinement types
  - No “false positive” mismatches on tensor shapes
  - No automated theorem proving is necessary
- Relies on the observation that
  tensor shapes are “not very dynamic” in many typical cases
  - i.e., typical programs do not use operations where
    the resulting shape cannot be obtained before actual tensor computation

Limitations

- Cannot verify properties that are essentially unable to be tested by evaluation
  - E.g., essentially universally quantified properties
  - Hence not very suitable for checking ***libraries***
- Handling tensors of shapes available only at runtime is feasible, but it does not totally prevent runtime failures
  - It is achieved simply by generating code ***at runtime*** and run it
  - ***Still, failures happen immediately when generating code***, not during the execution of generated code

Comparison to Major Similar Work

- Operationally, our approach resembles ***LMS-Verify*** [Amin & Rompf 2017]
  - But, our work has:
    - ***language-level*** safety guarantee
    - type-guided assertion insertion
  - Tracking shapes by types is a key ingredient

Comparison to Major Similar Work

- Operationally, our approach resembles ***LMS-Verify*** [Amin & Rompf 2017]
  - But, our work has:
    - ***language-level*** safety guarantee
    - type-guided assertion insertion
  - Tracking shapes by types is a key ingredient
- ***GraTen*** [Hattori, Sato, & Kobayashi 2024]
  - Best-effort inference based on refinement types
  - Fall-back assertions are at ***runtime***
  - Inspires our work as to assertion insertion and the support of broadcasting

Comparison to Major Similar Work

- Operationally, our approach resembles ***LMS-Verify*** [Amin & Rompf 2017]
  - But, our work has:
    - ***language-level*** safety guarantee
    - type-guided assertion insertion
  - Tracking shapes by types is a key ingredient

- ***GraTen*** [Hattori, Sato, & Kobayashi 2024]
  - Best-effort inference based on refinement types
  - Fall-back assertions are at ***runtime***
  - Inspires our work as to assertion insertion and the support of broadcasting

- ***Idris 2*** [Brady 2021]
  - Has somewhat staging-like distinction built upon ***QTT*** [Atkey 2018]
  - Compile time ≈ ***multiplicity*** 0 ?
  - Interoperability might be worth investigating

Conclusion

- Gave a formalization of ***compile-time*** tensor shape checking based on ***staged computation***,
  and proved its metatheoretic safety properties
  - Runtime safety is guaranteed once code generation succeeds
  - Major motivation: affinity with ***continuous*** software development
- Also proposed:
  - A non-staged surface language called ***Horsea***
  - An (intentionally plain) inference algorithm of implicit arguments
- Implemented a prototype type-checker in Haskell
  and made it publicly available:
  - https://github.com/gfngfn/Horsea