Compile-Time Tensor Shape Checking via Staged Shape-Dependent Types (Artifact)

Takashi Suwa
Kyoto University, Japan
Imiron Co., Ltd., Tokyo, Japan

Abstract

Based on the method proposed in the corresponding paper, we implemented a prototype type-checker (and a simple interpreter) for compile-time tensor shape checking. we ported a number of DNN-related example programs offered by `ocaml-torch` to our language and checked them by using the prototype implementation. The results indicated that our method can be considered effective enough to accommodate realistic tensor-handling programs.

2012 ACM Subject Classification Software and its engineering → Functional languages; Software and its engineering → Software verification and validation; Theory of computation → Type structures

Keywords and phrases Metaprogramming, Staged computation, Dependent types, Refinement types, Tensor shape checking

Digital Object Identifier 10.4230/DARTS.12.1.14

Funding This work is partially supported by JSPS KAKENHI Grant Numbers 20H00582 and 26H02493, Japan.

Acknowledgements The authors wish to thank the anonymous reviewers and (past) members of our laboratory for various fruitful comments and feedbacks. We are also grateful to Shivankur Gupta for implementing a basic code-generating backend for OCaml during his internship.

Related Article Takashi Suwa and Atsushi Igarashi, "Compile-Time Tensor Shape Checking via Staged Shape-Dependent Types", in 40th European Conference on Object-Oriented Programming (ECOOP 2026), LIPIcs, Vol. 372, pp. 28:1–28:31, 2026. `https://doi.org/10.4230/LIPIcs.ECOOP.2026.28`

Related Conference 40th European Conference on Object-Oriented Programming (ECOOP 2026), June 29–July 3, 2026, Brussels, Belgium

Evaluation Policy The artifact has been evaluated as described in the ECOOP 2026 Call for Artifacts and the ACM Artifact Review and Badging Policy.

1 Scope

Based on the method proposed in the corresponding paper, we implemented a prototype type-checker (and a simple interpreter) for compile-time tensor shape checking. Our purpose of implementing this tool was the following:

A. Because our method relies on the observation that tensor shapes in typical tensor-handling programs are "not very dynamic" (i.e., that typical programs contain a limited number of operations where the shape of resulting tensors can be determined only at runtime), it was unclear whether our method sufficiently accommodates realistic tensor-handling programs. By porting example programs offered by the repository of `ocaml-torch` [1] into our language and checking them by using the prototype implementation, we demonstrate that our language is expressive enough to support realistic programs (i.e., justify the claim around the fourth item of the contribution list on page 6 and Section 7 of the paper).

B. Although this was not mentioned in the submitted version of our paper, guided by the feedbacks by the anonymous reviewers, we would like to display a table that indicates how many implicit parameters are reconstructed by our inference algorithm for the ported example programs. We

Table 1 Properties of example programs and the inference results of implicit arguments

program	total	inferred	annot.
`char_rnn/char_rnn.hrs`	39	37	12
`gan/mnist_cgan.hrs`	59	55	5
`gan/mnist_dcgan.hrs`	112	106	4
`gan/mnist_gan.hrs`	51	47	4
`jit/load_and_run.hrs`	5	3	0
`min-gpt/mingpt.hrs`	108	96	17
`mnist/conv.hrs`	28	25	3
`mnist/linear.hrs`	20	20	1
`pretrained/finetuning.hrs`	45	35	6
`pretrained/predict.hrs`	8	5	0

use the prototype implementation to create such a table. This will quantitatively strengthen the claim that our inference algorithm is fairly effective for realistic uses, in contrast to it concise formalization.

C. While the core language of our method is proven to be type-safe by metatheoretic discussions (see Section 3.2 of the paper for details), its extension with implicit arguments and the non-staged surface language have yet to establish a solid mathematical guarantee (although they are heavily inspired by existing methods that fulfill type-safety properties). By preparing various concrete example programs and feeding them to the prototype implementation, we demonstrate that the extended features indeed work fine for the programs.

Table 1 in the present document shows the obtained results for the goal **B**. The columns "total" and "inferred" display the total number of implicit arguments in each program and the number of successfully inferred ones among those arguments, respectively. Those that cannot be inferred are manually specified in the programs (e.g., `char_rnn/char_rnn.hrs` contains $39 - 37 = 2$ manually specified implicit arguments, and the other 37 are appropriately reconstructed by the type-checker). The column "annot" shows the number of shape-related descriptions contained in type annotations in each program. We attached this table to the camera-ready version of our paper[1].

2 Content

2.1 Artifact package

The artifact package includes the following:

- `horsea-docker-image.tar`: The main Docker image.
- `README-to-start-container.md`: A small instruction about how to start the Docker image; the contents of this file overlaps the present document.
- `horsea/`: The codebase of the artifact; its contents are identical to those of `/horsea/` in the Docker image.

[1] In particular, after the author response, we have found that all the implicit arguments in `mnist/linear.hrs` can be reconstructed by slightly refining the type of `Tensor.cross_entropy_for_logits`. We thereby greatly updated Section 7 of the paper.

2.2 Docker image

The Docker image includes the following:

1. The implementation of a language processor consisting of the following:
   a. The non-staged surface language
      - **Type of artifact**: code
      - **Format**: Haskell source code
      - **Location in the container**: `/horsea/src/Surface/`
      - Performs binding-time analysis (BTA) to reconstruct stages. Resulting staged programs will be passed to **b** internally.
      - Important modules (paths are relative to `/horsea/src/`):
        - `Surface/Syntax.hs`: Defines the source syntax, i.e., the form of parsed ASTs.
        - `Surface/Parser.hs`: Parses programs in Horsea.
        - `Surface/BindingTime/Analyzer.hs`: The core part of the BTA algorithm.
        - `Surface/Entrypoint.hs`: The entrypoint of this component; The CLI component **c** calls a function provided by this module.
   b. The staged source language $\lambda^{\langle\rangle\{\}\{\}}$ with implicit parameters/arguments
      - **Type of artifact**: code
      - **Format**: Haskell source code
      - **Location in the container**: `/horsea/src/Staged/`
      - The type-checker elaborates programs in $\lambda^{\langle\rangle\{\}\{\}}$ to $\lambda^{\langle\rangle\{\}}$, the core language of our theory, by inserting casts and inferring omitted implicit arguments in a type-guided manner. As well as executing this type-checker via Horsea, users can write manually staged programs and pass them to this component directly. There is also a naïve back-end interpreter of $\lambda^{\langle\rangle\{\}}$ that performs compile-time computation for code generation (and possibly runtime computation afterwards).
      - Important modules (paths are relative to `/horsea/src/`):
        - `Staged/SrcSyntax.hs`: Defines the source syntax, i.e., the form of parsed ASTs.
        - `Staged/Parser.hs`: Parses programs in $\lambda^{\langle\rangle\{\}\{\}}$, i.e., those staged manually.
        - `Staged/Typechecker.hs`: The core part of the bidirectional type-checking algorithm.
        - `Staged/Syntax.hs`: Defines the internal syntax, i.e., the form of ASTs obtained by the type-guided elaboration.
        - `Staged/BuiltIn/Definitions.hs`: Defines the semantics of built-in functions.
        - `Staged/Evaluator.hs`: A naïve interpreter of $\lambda^{\langle\rangle\{\}}$.
        - `Staged/Entrypoint.hs`: The entrypoint of this component. The surface component **b** or the CLI component **c** calls a function provided by this module.
   c. The CLI interface
      - **Type of artifact**: code
      - **Format**: Haskell source code
      - **Location in the container**: `/horsea/app/Main.hs`
      - Provides the following subcommands:
        - `cabal run horsea - surface` ⟨*HorseaFile*⟩ ⟨*Options*⟩: Invokes **a**.
        - `cabal run horsea - staged` ⟨*StagedFile*⟩ ⟨*Options*⟩: Invokes **b**.
2. A stub file
   - **Type of artifact**: data (source code of the *implemented* languages)
   - **Format**: $\lambda^{\langle\rangle\{\}\{\}}$ source code for modules (`.lbam`)
   - **Location in the container**: `/horsea/stub.lbam`

   - Defines built-in functions for $\lambda^{\langle\rangle(|)\{\}}$, which are also indirectly available in Horsea. Each binding in this file must be consistent with the corresponding definition of semantics in `/horsea/src/Staged/BuiltIn/Definitions.hs`.
3. A number of example programs ported from `ocaml-torch` [1] to Horsea or to $\lambda^{\langle\rangle(|)\{\}}$ directly.
   - **Type of artifact**: data (source code of the *implemented* languages)
   - **Format**: Horsea source code (`.hrs`) and $\lambda^{\langle\rangle(|)\{\}}$ source code (`.lba`)
   - **Location in the container**: `/horsea/examples/ocaml-torch/`
   - These programs are used for the goals **A** and **B** mentioned above.
4. Small example programs used for integration tests.
   - **Type of artifact**: data (source code of the *implemented* languages)
   - **Format**: Horsea source code (`.hrs`) and $\lambda^{\langle\rangle(|)\{\}}$ source code (`.lba`)
   - **Location in the container**: `/horsea/examples/{small,failure}/`
   - Programs in `small/` are intended to be well-typed while those in `failure/` must be ill-typed. These programs are used for the goal **C**.

2.3 Reusability

The codebase and benchmarks are open-source; they are available at: `https://github.com/gfngfn/Horsea`. How to build, install, and run the implementation is concisely documented in the `README.md` file of the repository. Although the language specification has been only partially documented in `docs/syntax-staged.md`, we believe that our paper provides sufficient explanation for users to understand the syntax, the type system, and some other features, and that one can use tens of example programs in the directory `examples/` as references.

3 Getting

The following command loads the Docker image:

```
$ docker load < horsea-docker-image.tar
```

If the Docker image has been successfully loaded, the result of the following will include a line starting with "`horsea`":

```
$ docker images
```

Then, one can start the Docker image by invoking the following. This will start `/bin/bash` inside the container:

```
$ docker run -it horsea
```

4 Platforms

The artifact is functional at least under the following environment:

- OS: `Darwin` (kernel version: 24.6.0; macOS Sequoia 15.7.1)
- CPU: `arm64` (Apple M4)
- Memory: 24 GB
- Storage: 994.66 GB

We also suppose that the requirement of platforms is basically mild; any environments that accommodate GHC 9.4.8 [2], Cabal 3, and depended packages would work fine.

5 License

The artifact is available under the 3-Clause BSD License.

6 MD5 sum of the artifact

0b330dbaf7d9a69dcfa7ad3aa81bbdf5

7 Size of the artifact

1155403842 bytes ≈ 1.076 GiB

A How to run the artifact inside the Docker container

First, make sure that the current directory is `/horsea/` in the Docker container.

The following checks that all the example programs be verified as intended (i.e., that the prototype type-checker fulfills the goals **A** and **C**):

```
root@xxxxxxxxxxxx:/horsea# ./scripts/run-integration-tests.sh
```

If the check is successful, the above command finally prints the following line on stdout:

```
All tests have passed.
```

Also, the following displays the statistics about checking each programs in `examples/ocaml-torch/` (i.e., generates Table 1), satisfying the goal **B**:

```
root@xxxxxxxxxxxx:/horsea# ./scripts/get-stats.sh
```

References

1 Jane Street Capital. ocaml-torch. `https://github.com/janestreet/torch`, 2023. Accessed: 2025-02-27.

2 GHC Team. Glasgow Haskell Compiler. `https://www.haskell.org/ghc/`. Accessed: 2025-03-22.