Optimising Density Computations in Probabilistic Programs via Automatic Loop Vectorisation

SANGHO LIM*, KAIST, Korea
HYOUNGJIN LIM*, KAIST, Korea
WONYEOL LEE, POSTECH, Korea
XAVIER RIVAL, INRIA Paris, CNRS, ENS, and PSL University, France
HONGSEOK YANG, KAIST, Korea

Probabilistic programming languages (PPLs) are a popular tool for high-level modelling across many fields. They provide a range of algorithms for probabilistic inference, which analyse models by learning their parameters from a dataset or estimating their posterior distributions. However, probabilistic inference is known to be very costly. One of the bottlenecks of probabilistic inference stems from the iteration over entries of a large dataset or a long series of random samples. Vectorisation can mitigate this cost, but manual vectorisation is error-prone, and existing automatic techniques are often ad-hoc and limited, unable to handle general repetition structures, such as nested loops and loops with data-dependent control flow, without significant user intervention. To address this bottleneck, we propose a sound and effective method for automatically vectorising loops in probabilistic programs. Our method achieves high throughput using speculative parallel execution of loop iterations, while preserving the semantics of the original loop through a fixed-point check. We formalise our method as a translation from an imperative PPL into a lower-level target language with primitives geared towards vectorisation. We implemented our method for the Pyro PPL and evaluated it on a range of probabilistic models. Our experiments show significant performance gains against an existing vectorisation baseline, achieving 1.1–6× speedups and reducing GPU memory usage in many cases. Unlike the baseline, which is limited to a subset of models, our method effectively handled all the tested models.

CCS Concepts: • **Mathematics of computing → Probabilistic inference problems**; • **Computing methodologies → Parallel algorithms**; **Vector / streaming algorithms**.

Additional Key Words and Phrases: probabilistic programming, loop vectorisation, loop parallelism

ACM Reference Format:
Sangho Lim, Hyoungjin Lim, Wonyeol Lee, Xavier Rival, and Hongseok Yang. 2026. Optimising Density Computations in Probabilistic Programs via Automatic Loop Vectorisation. *Proc. ACM Program. Lang.* 10, POPL, Article 0 (January 2026), 31 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Over the last two decades, probabilistic programming languages (PPLs) have emerged as important tools for expressing and reasoning about models in machine learning, statistics, and computational science [1, 9, 17, 25, 51, 62, 63]. They are used to analyse datasets from epidemiology [52, 70] and ecology [33, 53] to phylogenetics [57] and robotics [16, 32]. PPLs enable users to model probabilistic phenomena naturally using primitives for probabilistic sampling and conditioning, as well as features

*Both authors contributed equally to this paper.

This work was supported by the National Research Foundation of Korea(NRF) grant funded by the Korean Government(MSIT) (No. RS-2023-00279680).
Authors' Contact Information: Sangho Lim, lim.sang@kaist.ac.kr, KAIST, Korea; Hyoungjin Lim, lmkmkr@kaist.ac.kr, KAIST, Korea; Wonyeol Lee, wonyeol.lee@postech.ac.kr, POSTECH, Korea; Xavier Rival, rival@di.ens.fr, INRIA Paris, CNRS, ENS, and PSL University, France; Hongseok Yang, hongseok.yang@kaist.ac.kr, KAIST, Korea.

from general-purpose programming languages, such as loops, branches, and more recently, trainable parameters (e.g., neural networks). These models are then analysed using generic inference engines of PPLs, which approximate posterior distributions or learn model parameters by performing algorithms from Bayesian statistics and deep learning, such as Hamiltonian Monte Carlo (HMC) [24, 42], Stochastic Variational Inference (SVI) [29], and Stochastic Gradient Descent (SGD) [34, 55].

This paper aims to reduce the high computational cost of PPL inference engines, a challenge that persists despite advances in statistical algorithms and compilation. Our work focuses on scenarios requiring repeated evaluation of probabilistic programs during parameter learning or posterior inference, particularly those involving intensive computations of model densities and their gradients arising from repetition structures, often implemented as for-loops. We develop a sound automatic method for optimising such loops through tensor-based vectorisation, thereby enabling the underlying inference engines to analyse these models more efficiently.

Models with repetition structures are common in practice. Graphical models from statistics even employs a special notation, called *plate*, to compactly describe such structures. The simplest repetition structure is a loop over a dataset where each iteration is independent of the others in the sense that it does not have data dependency from the other iterations. Our optimisation allows the inference engines to process such loops in parallel using tensors, yielding significant benefits for medium-to-large datasets. More challenging examples come from sequence models and hierarchical models, such as Bayesian hierarchical models [4, 20] and hidden Markov models [72], where repetition structures are implemented as nested loops, or they lack independence and are coded as loops whose iterations are data-dependent on the previous ones. Even in these complex scenarios, our optimisation enables the inference engines to identify the available parallelism and exploit it through tensor-based vectorisation, achieving significant speedups in posterior inference and parameter learning.

Speeding up PPL inference engines using tensor-based vectorisation is a known idea in the PPL community, supported by a range of existing primitives, such as `vmap` [13], `vectorized_map` [2], `vectorized_markov` [46], and the `MarkovChain` distribution [23]. However, these primitives fall short of fully automatic vectorisation of the general repetition structures that we target. Concretely, they require users to write repetition structures explicitly in vectorised form, or to provide vectorisation-related information, e.g., dependency information in the repetition structures. This task is both cumbersome and error-prone; for example, manually vectorising the loop in Fig. 1 requires care due to inter-iteration data dependencies, as detailed in §2. Moreover, these primitives are limited, and cannot handle nested repetition structures or dynamically varying dependencies arising from internal random choices; they often support only repetition without data dependencies. In contrast, our vectorisation is fully automatic, requires no user input, and correctly handles general repetition structures, including nested ones and those with complex, randomly-varying dependencies.

When viewed as a code transformer, our optimisation can be understood as a *non-standard* loop vectoriser. Given a model with a repetition structure expressed as a for-loop over $\{0, 1, \ldots, n-1\}$, our optimisation transforms the loop to parallelism-friendly tensor-based vectorised code. The vectorised code here is subtle, which goes beyond the simple standard idea of executing all the n iterations of the original loop in parallel. To handle the case that the iterations of the loop are data-dependent, the code implements two novel ideas: (i) *speculative execution* of the loop body, and (ii) the use of a *fixed-point check* for testing the validity of speculation.

(i) The code runs all the n iterations of the original loop in parallel, but some iterations are *speculative* in the sense that their starting states may differ from those of the corresponding iterations in the original loop. This speculation lets the code avoid waiting until the starting states of all the iterations of the original loop are known. But the outcomes of these speculative computations may diverge from those of the corresponding iterations of the original loop.

(ii) To address this divergence issue, the code repeats the parallel (and partly speculative) execution of all the n iterations of the loop multiple times, and terminates this repetition when it finds out that all the speculative executions in the current repetition are correct, i.e., they correspond to the n iterations of the original loop. The termination mechanism here employs a *fixed-point check* to detect the correctness of speculation, and it is guaranteed to stop the repetition within at most n iterations (due to a particular type of speculation we use).

For instance, as we explain in §3, our optimisation transforms the loop of Fig. 1 into vectorised code with a fixed-point check, which ensures that the parallel execution of the original loop is repeated just twice. If the reader is familiar with abstract interpretation, a good intuition is that the vectorised code implements an executable version of the collecting semantics from the abstract interpretation literature; an execution in the collecting semantics corresponds to a set of executions in the standard semantics, and loops in the collecting semantics are interpreted in terms of fixed-points.

When a model with a small degree of data dependency (e.g., a low-order Markov model) is expressed in a PPL that supports a powerful tensor framework, tensor broadcasting, and efficient parallel tensor operations, our optimisation can vectorise the computation of the model's density. The vectorised model runs more efficiently than the original one, thanks to parallelism and early termination enabled by our fixed-point check. Moreover, our vectorisation also accelerates gradient computations over the model's density through the automatic differentiation provided by the PPL's backend. In this case, automatic differentiation operates on the vectorised computation graph of the density, thereby computing gradients in a vectorised manner as well. Such models and PPLs are common, and the resulting speedup directly improves the performance of PPL inference engines that repeatedly calculate or differentiate the model's density, such as Stochastic Variational Inference (SVI) and Maximum a Posteriori (MAP) estimation.

We prove the correctness of our optimisation using the formalism of expressing and analysing tensor-based vectorised computations, which we also develop in the paper. To evaluate the effectiveness of our optimisation, we implemented it for the Pyro PPL, and applied it to posterior inference and parameter learning by SVI and MAP estimation as well as to Markov Chain Monte Carlo (MCMC) sampling.[1] Our experiments on eight Pyro models show that our optimisation significantly enhances inference performance, achieving 1.1–6× speedups over an existing vectorisation baseline, and reducing GPU memory usage to 14–80% for seven out of eight models, with one outlier exhibiting the increase to 109–112%. The vectorisation baseline fails on some of these eight models, while our optimisation was successfully applied to all the models.

We summarise the contributions of this paper below:

(1) We propose a language with specialised primitives that enables tensor-based vectorised execution of probabilistic programs. It is designed to facilitate vectorisation of loops in PPLs and support a broader class of models with control flow and nested structures (§3 and §4).
(2) We present a transformation that converts standard probabilistic programs into our proposed language for vectorisation. We prove the correctness of this transformation (§3 and §4).
(3) We implemented our transformation on top of the Pyro PPL, and evaluated it on several families of models, demonstrating that the transformation preserves program semantics while significantly improving execution time and in some cases dramatically reducing memory usage especially when the models have short-range data dependencies. (§5 and §6).

We clarify that our work focuses on optimising density and gradient computations in probabilistic programs, but does not address the sample-generation cost in sampling-based inference algorithms.

[1]Our implementation is available at https://github.com/Lim-Sangho/auto-vectorise-ppl.public.

$$HMM = \left[\begin{array}{l} x := 0.0; y := 0.0; \\ \textbf{for } t \textbf{ in range}(10) \textbf{ do } \{ \\ \quad x := \textbf{fetch}([(\text{“}\mathbf{z}\text{”}, t)]); \\ \quad \textbf{score}(\log p_{\mathcal{N}}(x; f(y), 1.0)); \\ \quad \textbf{score}(\log p_{\mathcal{N}}(o(t); g(x), 1.0)); \\ \quad y := x \ \} \end{array}\right], \quad HMM' = \left[\begin{array}{l} x := 0.0; y := 0.0; \\ \textbf{for } t \textbf{ in range}(10) \textbf{ do } \{ \\ \quad x := \textbf{sample}([(\text{“}\mathbf{z}\text{”}, t)], \mathcal{N}(f(y), 1.0)); \\ \quad \textbf{observe}(\mathcal{N}(g(x), 1.0), o(t)); \\ \quad y := x \ \} \end{array}\right]$$

Fig. 1. Example programs describing a hidden Markov model. Here, *HMM* is written in the language presented in §2, and *HMM′* is written in a typical probabilistic programming language.

2 A Simple Programming Language for Unnormalised Probability Densities

This paper is concerned with a class of PPL inference engines that view probabilistic programs as *evaluators* of unnormalised posterior densities, instead of *generators* of weighted samples, and repeatedly run these evaluators during inference or parameter learning. This class includes Stochastic Variational Inference (SVI), Hamiltonian Monte Carlo (HMC), Maximum Likelihood Estimation (MLE), and Maximum a Posteriori (MAP) estimation, forming the core of popular PPLs such as Pyro and Stan. Our goal is to optimise the density evaluators expressed by probabilistic programs.

In this section, we formalise this density-evaluator view of probabilistic programs. We start with an informal overview (§2.1), and then formally define a simple programming language, where a program assumes a fixed collection of random variables, takes the values of these random variables as inputs, and then returns a real number, called total score, as an output (§2.2 and §2.3).

2.1 Overview of a Score-Computing Language

In the imperative language to be introduced in §2.2, programs are intended to compute the unnormalised (posterior) probability densities of some probabilistic models, where being unnormalised means the integrals of the densities at all inputs are not 1 (which comes mostly from conditioning in those models). Their inputs are the values of the random variables in the models, and their outputs are the logarithms of the unnormalised densities at the given input values.

For instance, the program *HMM* in Fig. 1 describes the unnormalised posterior density of a hidden Markov model (HMM), and corresponds to *HMM′* in the same figure written with the more traditional **sample** and **observe** constructs. The program defines a probabilistic model over random variables $\mathbf{z}_t$ and $\mathbf{o}_t$ for $t \in \{0, \ldots, 9\}$, where $\mathbf{z}_t$ (which is referred to by the notation $[(\text{“}\mathbf{z}\text{”}, t)]$ in *HMM* with the string “$\mathbf{z}$”) denotes the hidden state of a stochastic machine at step t, and $\mathbf{o}_t$ (which is used only implicitly by the second **score** command in *HMM*) means the observation on the state $\mathbf{z}_t$ made at step t. The probability density of the model is

$$p_{HMM}(\mathbf{z}_0, \ldots, \mathbf{z}_9, \mathbf{o}_0, \ldots, \mathbf{o}_9) = \textstyle\prod_{t=0}^{9} \Big(p_{\mathcal{N}}(\mathbf{z}_t; f(\mathbf{z}_{t-1}), 1.0) \cdot p_{\mathcal{N}}(\mathbf{o}_t; g(\mathbf{z}_t), 1.0) \Big),$$

where f, g are the functions in *HMM*, $p_{\mathcal{N}}$ the density of the normal distribution, and $\mathbf{z}_{-1}$ the constant 0.0. The program conditions this density with observations $\mathbf{o}_t = o(t)$ for all t, where the $o(t)$ denote fixed observed values. Then, it computes the following quantity using the **score** command:

$$\log p_{HMM}(\mathbf{z}_0, \ldots, \mathbf{z}_9, o(0), \ldots, o(9)) = \textstyle\sum_{t=0}^{9} \Big(\log p_{\mathcal{N}}(\mathbf{z}_t; f(\mathbf{z}_{t-1}), 1.0) + \log p_{\mathcal{N}}(o(t); g(\mathbf{z}_t), 1.0) \Big), \quad (1)$$

where the values of $\mathbf{z}_t$ are given to the program as inputs. In each iteration of the for-loop, the program reads the input value of $\mathbf{z}_t$ using the **fetch** construct and computes the two terms in the summand of Eq. (1) using the **score** construct. The outcomes of **score** get accumulated, and the final sum is returned as the output: the logarithm of the unnormalised density in Eq. (1).

In the rest of this section, we formally describe the syntax and semantics of this language of score-computing programs, which will be used to define our optimisation and prove its correctness.

Syntax:

$$
\begin{array}{rrcl}
\textit{Integer Expressions} & Z & ::= & n \mid x^{\mathsf{int}} \mid \mathrm{op}_i(Z_1, \ldots, Z_k, E_1, \ldots, E_m) \\
\textit{Real Expressions} & E & ::= & r \mid x^{\mathsf{real}} \mid \mathrm{op}_r(Z_1, \ldots, Z_k, E_1, \ldots, E_m) \\
\textit{Index Expressions} & I & ::= & [(\alpha_1, Z_1); \ldots; (\alpha_k, Z_k)] \quad (\text{for distinct } \alpha_i\text{'s}) \\
\textit{Commands} & C & ::= & \mathbf{score}(E) \mid x^{\mathsf{real}} := \mathbf{fetch}(I) \mid \mathbf{skip} \mid x^{\mathsf{int}} := Z \mid x^{\mathsf{real}} := E \\
& & & \mid C_1; C_2 \mid \mathbf{ifz}\ Z\ C_1\ C_2 \mid \mathbf{for}\ x^{\mathsf{int}}\ \mathbf{in}\ \mathbf{range}(n)\ \mathbf{do}\ C \quad (\text{for } n \geq 1)
\end{array}
$$

Semantics:

$$
\begin{array}{rr}
[\![n]\!]\sigma = n & [\![x^{\mathsf{int}}]\!]\sigma = \sigma(x^{\mathsf{int}}) \\
[\![\mathrm{op}_i(Z_1, \ldots, Z_k, E_1, \ldots, E_m)]\!]\sigma = [\![\mathrm{op}_i]\!]_a([\![Z_1]\!]\sigma, \ldots, [\![Z_k]\!]\sigma, [\![E_1]\!]\sigma, \ldots, [\![E_m]\!]\sigma) & \\
[\![r]\!]\sigma = r & [\![x^{\mathsf{real}}]\!]\sigma = \sigma(x^{\mathsf{real}}) \\
[\![\mathrm{op}_r(Z_1, \ldots, Z_k, E_1, \ldots, E_m)]\!]\sigma = [\![\mathrm{op}_r]\!]_a([\![Z_1]\!]\sigma, \ldots, [\![Z_k]\!]\sigma, [\![E_1]\!]\sigma, \ldots, [\![E_m]\!]\sigma) & \\
[\![[(\alpha_1, Z_1); \ldots; (\alpha_k, Z_k)]]\!]\sigma = [(\alpha_1, [\![Z_1]\!]\sigma); \ldots; (\alpha_k, [\![Z_k]\!]\sigma)] &
\end{array}
$$

$$
\frac{}{(x^{\mathsf{real}} := \mathbf{fetch}(I)), D, \sigma \Downarrow \sigma[x^{\mathsf{real}} \mapsto D([\![I]\!]\sigma)], 0.0} \qquad \frac{}{\mathbf{score}(E), D, \sigma \Downarrow \sigma, [\![E]\!]\sigma}
$$

$$
\frac{}{(x^{\mathsf{int}} := Z), D, \sigma \Downarrow \sigma[x^{\mathsf{int}} \mapsto [\![Z]\!]\sigma], 0.0} \qquad \frac{}{(x^{\mathsf{real}} := E), D, \sigma \Downarrow \sigma[x^{\mathsf{real}} \mapsto [\![E]\!]\sigma], 0.0}
$$

$$
\frac{C_1, D, \sigma \Downarrow \sigma', r_1 \quad C_2, D, \sigma' \Downarrow \sigma'', r_2}{(C_1; C_2), D, \sigma \Downarrow \sigma'', (r_1 + r_2)} \qquad \frac{[\![Z]\!]\sigma = 0 \quad C_1, D, \sigma \Downarrow \sigma', r}{(\mathbf{ifz}\ Z\ C_1\ C_2), D, \sigma \Downarrow \sigma', r} \qquad \frac{[\![Z]\!]\sigma \neq 0 \quad C_2, D, \sigma \Downarrow \sigma', r}{(\mathbf{ifz}\ Z\ C_1\ C_2), D, \sigma \Downarrow \sigma', r}
$$

$$
\frac{}{\mathbf{skip}, D, \sigma \Downarrow \sigma, 0.0} \qquad \frac{\sigma_0 = \sigma \quad C, D, \sigma_k[x \mapsto k] \Downarrow \sigma_{k+1}, r_{k+1} \text{ for all } k \in \{0, \ldots, n-1\}}{(\mathbf{for}\ x\ \mathbf{in}\ \mathbf{range}(n)\ \mathbf{do}\ C), D, \sigma \Downarrow \sigma_n, \sum_{k=1}^{n} r_k}
$$

Fig. 2. Syntax and semantics of the score-computing language.

2.2 Syntax

Let $\mathbb{S}, \mathbb{Z}, \mathbb{R}$, *Var* be the sets of strings, integers, real numbers, and variables, respectively. We use the following symbols to range over elements in these sets: $\alpha, \beta \in \mathbb{S}$, $n, m \in \mathbb{Z}$, $r \in \mathbb{R}$, and $x, y, z \in \mathit{Var}$. Each variable $x \in \mathit{Var}$ has a type $\tau \in \{\mathsf{int}, \mathsf{real}\}$. We often write x^τ to denote that x has type τ.

The syntax of the score-computing language is given in Fig. 2, based on the above notations. The language has three types of expressions, which denote state-dependent integers, reals, and *indices*. By indices, we mean finite sequences of string-integer pairs that do not use the same string more than once, that is, elements i in the following set:

$$
i \in \mathit{Index} = \left\{ [(\alpha_1, m_1); \ldots; (\alpha_k, m_k)] \in (\mathbb{S} \times \mathbb{Z})^* \ : \ \alpha_j\text{'s are distinct for } j \in \{1, \ldots, k\}, k \geq 0 \right\}.
$$

For instance, when evaluated at a state, an index expression $[(\alpha_1, Z_1); \ldots; (\alpha_k, Z_k)]$ returns the index $[(\alpha_1, m_1); \ldots; (\alpha_k, m_k)]$, i.e., the length-$k$ sequence of string-integer pairs (α_j, m_j), where m_j is the integer value of Z_j at the state. These indices serve as identifiers of random variables in the language. When a command in the language needs the value of a random variable, it uses the index for the variable as a key and looks up an entry at this key in a table, called *random database*, which stores the values of all the random variables. The grammars for the expressions are standard, except for the use of op_i and op_r, the former being an integer-returning operation with appropriate input types and the latter being a real-returning operation.

Commands in the language denote variants of state transformers, which take a random database and a starting state as inputs, and return a final state and a real number called *total score* as outputs. A random database D here refers to a map from all indices (i.e., finite sequences of string-integer pairs with no repetition of a string) to real numbers, i.e., $D \in \mathit{RDB} = [\mathit{Index} \to \mathbb{R}]$. It specifies the

values of random variables (identified by the indices). Thus, besides updating the state, a command C computes the total score of the given random database. When C represents a probabilistic model, the computed score is the logarithm of an unnormalised density of the model.

The command $\mathbf{score}(E)$ evaluates E in the current state, and returns the result as the total score, while keeping the state unchanged. The next command $x^{\mathsf{real}} := \mathbf{fetch}(I)$ reads the value at the index I in the random database, stores the read value in x^{real}, and returns the resulting state together with zero score. The other commands are the standard constructs of the imperative languages: **skip**, assignments, sequencing, conditionals, and for-loops. Regarding the state transformation, they have the standard meanings, except that the conditionals use an implicit zero check on their Z argument, so that their true branch C_1 gets executed if Z evaluates to zero, and the false branch C_2 is followed otherwise. Regarding the score computation, these commands implement the following two principles: (i) when a command C is run, it may execute the **score** command multiple times, and the results of these multiple invocations of **score** get added to become the final total score of C; (ii) none of the other atomic commands, such as assignments and **skip**, contributes to the total score. Note that the language is restricted in the sense that it does not have general while-loops. This restriction ensures that all commands in the language always terminate, and this termination property is used when we prove the correctness of our optimisation. We point out that despite this restriction, the language is expressive enough to allow a wide range of typical probabilistic models, as indicated by the eight models used in our experiments. Lifting this restriction is, however, an interesting and important future work. We write Com for the set of all commands in the language.

2.3 Semantics

Let State be the set of states defined by the type-preserving maps from variables to integers or reals:

$$\mathit{State} = \{\sigma \in [\mathit{Var} \to \mathbb{Z} \cup \mathbb{R}] \ : \ \sigma(x^{\mathsf{int}}) \in \mathbb{Z} \text{ and } \sigma(y^{\mathsf{real}}) \in \mathbb{R} \text{ for all variables } x^{\mathsf{int}} \text{ and } y^{\mathsf{real}}\}.$$

We interpret expressions as maps from states to the values of appropriate types: $[\![Z]\!] : \mathit{State} \to \mathbb{Z}$, $[\![E]\!] : \mathit{State} \to \mathbb{R}$, and $[\![I]\!] : \mathit{State} \to \mathit{Index}$. We assume that the semantics of integer-returning or real-returning atomic operations in the language are given, i.e., we have $[\![\mathrm{op}_i]\!]_a : \mathbb{Z}^k \times \mathbb{R}^m \to \mathbb{Z}$ and $[\![\mathrm{op}_r]\!]_a : \mathbb{Z}^k \times \mathbb{R}^m \to \mathbb{R}$ for all op_i and op_r taking k integer and m real arguments. Using this assumed semantics of atomic operations, we interpret expressions in the standard way (Fig. 2).

We define the meanings of commands using a big-step operational semantics, which specifies the relation $\Downarrow \subseteq (\mathit{Com} \times \mathit{RDB} \times \mathit{State}) \times (\mathit{State} \times \mathbb{R})$. Intuitively, $(C, D, \sigma \Downarrow \sigma', r)$ means that running C on the state σ under the random database D terminates successfully, producing the state σ' and the total score r. The score r is the logarithm of the unnormalised density at D according to a model that C implements. Since the language does not have general while-loops, and no commands get stuck or generate an error, every tuple (C, D, σ) of command, random database, and input state leads to a unique pair (σ', r) made of an output state and a total score. We define $\Downarrow$ in the standard manner using the inference-rule notation (Fig. 2).

3 Loop Vectorisation

Our optimisation operates on commands in the language of §2 by vectorising for-loops in those commands. In this section, we describe this loop vectorisation as a translation from the language to another language that supports vectorised operations while continuing to have the primitives for describing unnormalised densities. We informally explain and formally define this *target language* of the translation, where all the atomic commands are implicitly vectorised and new language constructs control various aspects of vectorisation (§3.1 and §3.2). The target language is closely related to the popular tensor-based programming languages, such as Python with PyTorch tensors, which our implementation is based on (see §5). After defining the target language, we describe

$$C_0 = \left[\begin{array}{l} x^{\mathsf{real}\dagger} := \mathbf{fetch}([(\text{“}\mathbf{z}\text{”}, t^{\mathsf{int}\dagger})]); \\ \mathbf{score}(\log p_N(x^{\mathsf{real}\dagger}; f(y^{\mathsf{real}\dagger}), 1.0)); \\ \mathbf{score}(\log p_N(o(t^{\mathsf{int}\dagger}); g(x^{\mathsf{real}\dagger}), 1.0)); \\ y^{\mathsf{real}\dagger} := x^{\mathsf{real}\dagger} \end{array}\right], \qquad C_1 = \left[\begin{array}{l} \mathbf{extend\_index}(\text{“}\mathbf{vec}\text{”}, 10)\ \{ \\ \quad t^{\mathsf{int}\dagger} := \mathbf{lookup\_index}(\text{“}\mathbf{vec}\text{”}); \\ \quad C_0\ \} \end{array}\right]$$

Fig. 3. Example commands in the target language.

the translation from the language of §2 to the target language, and explain the use of speculative computations and fixed-point check in the translated commands, which we mentioned in the introduction (§3.3). In the rest of the paper, we refer to the language of §2 as the *source language*.

3.1 Overview of a Target Language for Loop Vectorisation

The target language is an imperative language with a restricted form of recursion, and includes all the constructs from the source language. Its two main features are as follows.

First, variables and expressions in the target language have *lifted types* rather than the original types in the source language, where lifting means changing a type τ to the type of maps from *Index* to τ. Concretely, variables in the target language store (finite representations of) maps from indices to integers or reals, instead of just integers or reals. Similarly, expressions in the target language evaluate to maps from indices to integers, reals, or indices.

Second, commands in the target language cannot use indices explicitly in order to access entries in these maps. Instead, they maintain a *set of indices* implicitly, and these indices are then used when expressions and variables are accessed during execution. Note that an index set, instead of a single index, is maintained here. This means that a single execution of a command in the target language corresponds to multiple standard executions of the same command, one per each index in the index set, where these executions access different entries of the lifted variables using their indices. This execution model of the target language is similar to the SIMD (single instruction multiple data) model, and can also be understood as an executable special case of the collecting semantics in the abstract interpretation literature.

To help the reader to gain a high-level intuition about how this target language works, we consider two commands in the language: C_0 and C_1 in Fig. 3.

Example 1 (C_0 in Fig. 3)**.** The command C_0 corresponds to a variant of the loop body of *HMM* in Fig. 1, where all occurrences of variables are annotated with their types.

Lifted types and index sets. The annotations in C_0 use the *lifted types*, real† and int†, instead of real and int; these lifted types indicate that the variables store not real numbers or integers, but maps to them. For instance, the variable t can store the following map from indices to integers:

$$m_t = \lambda i \in \mathit{Index}.\ \text{if } \left(i = [(\text{“}\mathbf{vec}\text{”}, \ell)] +\!\!+ i' \text{ for some } \ell \in \{0, 1, \ldots, 9\} \text{ and } i'\right) \text{ then } \ell \text{ else } 0. \qquad (2)$$

Here $j +\!\!+ j'$ means the concatenation of indices j and j' which are viewed as finite sequences of string-integer pairs. For this m_t, we have $m_t([(\text{“}\mathbf{vec}\text{”}, \ell)]) = \ell$ for all $\ell \in \{0, \ldots, 9\}$.

Some entries of the maps stored in the variables t, x, y are read and updated during the execution of C_0. But which entries are read or updated at each line of C_0 is not specified explicitly. Instead, it is determined implicitly by an *index set* that is given in the beginning of the execution.

Concretely, assume that C_0 is run under the index set $A_{vec} = \{i_0, i_1, \ldots, i_9\}$ for $i_\ell = [(\text{“}\mathbf{vec}\text{”}, \ell)]$, as well as a random database D and a starting state σ of the target language such that $D([(\text{“}\mathbf{z}\text{”}, \ell)]) = z_\ell$, $\sigma(t) = m_t$, $\sigma(x) = m_x$, and $\sigma(y) = m_y$. Here z_ℓ is the value of the random variable $\mathbf{z}_\ell$ of *HMM* that we used in §2, m_t is the map in Eq. (2), and m_x, m_y are the maps defined by $m_x(i) = m_y(i) = 0.0$ for all i. In this setup, C_0 is run as follows.

First command. The first command $x^{\mathsf{real}^\dagger} := \mathbf{fetch}([(\text{“}\mathbf{z}\text{”}, t^{\mathsf{int}^\dagger})])$ reads the ten entries of the map m_t at $i_0, \ldots, i_9 \in A_{vec}$, and forms the ten keys $[(\text{“}\mathbf{z}\text{”}, m_t(i_0))] = [(\text{“}\mathbf{z}\text{”}, 0)], \ldots, [(\text{“}\mathbf{z}\text{”}, m_t(i_9))] = [(\text{“}\mathbf{z}\text{”}, 9)]$ for the random database D. Then, it accesses D using these keys, and updates the map m_x with the results of these accesses: for each index $i_\ell \in A_{vec}$, the value $D([(\text{“}\mathbf{z}\text{”}, \ell)]) = z_\ell$ is copied to all the entries of m_x whose indices have the form $i_\ell +\!\!+ i'$ for some i'. Note that the update here is *non-standard* in that the copy of the read value z_ℓ is not restricted to the i_ℓ-th entry of m_x. More precisely, the resulting state is

$$\sigma[x \mapsto m'_x] \quad \text{where } m'_x(i) = \text{if } \big(i = i_\ell +\!\!+ i' \text{ for some } \ell \in \{0, \ldots, 9\} \text{ and } i'\big) \text{ then } z_\ell \text{ else } m_x(i). \quad (3)$$

As we will explain in §4, this non-standard variable update models *tensor broadcasting* in PyTorch and other tensor libraries, and is one of the factors that let us have an effective tensor-based implementation of maps stored in variables. Except for the use of this update, $\sigma[x \mapsto m'_x]$ is precisely the state that we would obtain by vectorising the corresponding **fetch** command in the source language over A_{vec} in the usual sense, i.e., running the command component-wise over a vector of size ten whose components are accessed by indices $i_0, \ldots, i_9$.

The first command $x^{\mathsf{real}^\dagger} := \mathbf{fetch}([(\text{“}\mathbf{z}\text{”}, t^{\mathsf{int}^\dagger})])$ also returns scores for the ten indices in A_{vec}, which happen to be all zero in this case. The returned scores are packaged into the constant zero map from A_{vec} to reals, written as $[i_0 \mapsto 0.0, \ldots, i_9 \mapsto 0.0]$.

Remaining commands. The remaining three commands are executed in a similarly A_{vec}-vectorised way, and give rise to the following final state σ' and score map $T' : A_{vec} \to \mathbb{R}$:

$$\sigma' = \sigma[x \mapsto m'_x, y \mapsto m'_x], \qquad T'(i_\ell) = \log p_{\mathcal{N}}(z_\ell; f(0.0), 1.0) + \log p_{\mathcal{N}}(o(\ell); g(z_\ell), 1.0), \quad (4)$$

where m'_x is the map in Eq. (3). These outputs correspond to the combined outcomes of all the ten iterations of the loop in *HMM* in the source language, where for each $\ell \in \{0, \ldots, 9\}$, the ℓ-th iteration is run after t is set to ℓ and both x and y are set to 0.0.

Note that the sum of the scores in T' differs from r_{HMM}, the total score computed by *HMM* in the source language with random variables $\mathbf{z}_\ell$ fixed to the values z_ℓ (see Eq. (1)):

$$\begin{aligned} \textstyle\sum_{\ell=0}^{9} T'(i_\ell) &= \textstyle\sum_{\ell=0}^{9} \Big(\log p_{\mathcal{N}}(z_\ell; f(\boxed{0.0}), 1.0) + \log p_{\mathcal{N}}(o(\ell); g(z_\ell), 1.0)\Big) \\ &\neq \textstyle\sum_{\ell=0}^{9} \Big(\log p_{\mathcal{N}}(z_\ell; f(\boxed{z_{\ell-1}}), 1.0) + \log p_{\mathcal{N}}(o(\ell); g(z_\ell), 1.0)\Big) = r_{HMM}, \end{aligned} \quad (5)$$

where $z_{-1} = 0.0$. The different parts are highlighted with boxes. They originate from the fact that the execution of C_0 under A_{vec} is *speculative* about the starting value of the variable y; at each index $i_\ell \in A_{vec}$, the i_ℓ part of the execution uses 0.0 as the starting value of y, instead of $z_{\ell-1}$ used by *HMM*. This type of speculation over starting values of variables at each index is exploited in our loop vectorisation, as we will explain in §3.3. □

Example 2 (C_1 in Fig. 3)**.** The command C_1 extends C_0 by adding the **lookup_index** command and then wrapping the whole with the **extend_index** construct. The **extend_index** construct *locally changes* the index set given in the beginning of the execution, and in so doing, it alters the level of vectorisation of commands inside its scope temporarily.

Concretely, assume that C_1 is executed under the index set $A_{[]} = \{[]\}$ containing only the empty index (i.e., empty sequence) and the starting state σ_1 where $\sigma_1(t) = (\lambda i.\, 0)$ and $\sigma_1(x) = \sigma_1(y) = (\lambda i.\, 0.0)$ (i.e., t, x, y store constant-zero maps modulo return types). Also, assume that the execution is given the same random database D as in Example 1. In this setup, C_1 is run as follows.

Entry phase and body of extend_index. The **extend_index** construct in C_1 first *extends* the index [] in $A_{[]}$ by adding the string-integer pair (“**vec**”, ℓ) at the end for every $\ell \in \{0, \ldots, 9\}$. The updated index set is $A_{vec} = \{i_0, \ldots, i_9\}$ from Example 1.

$$
\begin{array}{rcl}
\textit{Lifted Integer Expr. } Z & ::= & n \mid x^{\mathsf{int}\dagger} \mid \mathrm{op}_i(Z_1, \ldots, Z_k, E_1, \ldots, E_m) \\
\textit{Lifted Real Expr. } E & ::= & r \mid x^{\mathsf{real}\dagger} \mid \mathrm{op}_r(Z_1, \ldots, Z_k, E_1, \ldots, E_m) \\
\textit{Lifted Index Expr. } I & ::= & [(\alpha_1, Z_1); \ldots; (\alpha_k, Z_k)] \quad (\text{for distinct } \alpha_i\text{'s}) \\
\textit{Commands } C & ::= & \mathbf{score}(E) \mid x^{\mathsf{real}\dagger} := \mathbf{fetch}(I) \mid \mathbf{skip} \mid x^{\mathsf{int}\dagger} := Z \mid x^{\mathsf{real}\dagger} := E \mid C_1; C_2 \\
& & \mid \mathbf{for}\ x^{\mathsf{int}\dagger}\ \mathbf{in}\ \mathbf{range}(n_+)\ \mathbf{do}\ C \mid \mathbf{ifz}\ Z\ C_1\ C_2 \mid \mathbf{loop\_fixpt\_noacc}(n_+)\ C \\
& & \mid \mathbf{extend\_index}(\alpha, n_+)\ C \mid x^{\mathsf{int}\dagger} := \mathbf{lookup\_index}(\alpha) \mid \mathbf{shift}(\alpha)
\end{array}
$$

Fig. 4. Syntax of the target language. n, m are non-negative integers, n_+ a positive integer, and α a string.

The **extend_index** construct then executes the commands in its scope under A_{vec}, D, and σ_1. The first command in the scope is $t^{\mathsf{int}\dagger} := \mathbf{lookup\_index}(\text{“}\mathbf{vec}\text{”})$. For every index $i_\ell \in A_{vec}$, it finds the value paired with “**vec**” in i_ℓ, and copies the value to all the entries of the map $\sigma_1(t)$ whose indices are of the form $i_\ell +\!\!+ i'$ for some i'. That is, the resulting state is $\sigma_1[t \mapsto m_t]$ where m_t is the map in Eq. (2), so that the state is the same as the starting state σ of C_0 from Example 1. Regarding the score map, the command does not perform anything special, and simply returns the zero map $[i_0 \mapsto 0.0, \ldots, i_9 \mapsto 0.0]$. The remaining command in the scope is C_0. It is run under A_{vec}, D, and σ, and results in the final state σ' and the score map T' from Example 1 (see Eq. (4)).

Exit phase of extend_index. The **extend_index** construct finally *restores* the original index set $A_{[]}$ so that the subsequent computation is run under $A_{[]}$. Moreover, the construct *post-processes* σ' and T' to σ'' and T'', which are then returned as the final state and score map. Intuitively, σ'' is obtained by making the result at the last index $i_9 \in A_{vec}$ the final outcome of the entire computation, and T'' is obtained by summing over all the entries in A_{vec}. Concretely, for every variable w, the postprocessing builds the new map $m''_w = \sigma''(w)$ from the old map $m'_w = \sigma'(w)$, by reading the i_9-th entry of m'_w and storing the read value at every index: $m''_w = (\lambda i.\, m'_w(i_9))$. The score map T'' is the singleton function mapping the unique index $[] \in A_{[]}$ to $\sum_{\ell=0}^{9} T'(i_\ell)$. This post-processing comes from our loop vectorisation; when the vectorised parallel execution of all the loop iterations is completed, we need to pick only the final state computed by the last iteration but we have to accumulate the scores computed by all the iterations. The details will appear in §3.3. □

3.2 Syntax of the Target Language

Fig. 4 shows the syntax of the target language, which is obtained from that of the source language by annotating variables with lifted types real† and int†, instead of real and int, and including four new constructs, one for a special type of looping and three for controlling vectorisation. Commands in the target language operate on states with the lifted variables, i.e., variables that store maps from indices to reals or integers. As input, they receive a random database D, a state σ of lifted variables, and a finite set A of indices. Then, they output a final state σ' and a score map T' that assigns a score to each index $i \in A$. We write Com_{tgt} for the set of commands in the target language.

Let us go through the cases of a command C in the grammar and explain its behaviour assuming that C is run under (D, σ, A).

Existing commands. If the command C is one of the first five cases in the grammar, i.e., an atomic command that also appears in the source language (modulo type annotations of variables), then its execution in the target language corresponds to the *vectorised parallel execution* of the same command in the source language over indices in A. For each index $i \in A$, C is run according to the semantics of the source language, except that all variable reads are now on the i entries of the maps stored in those variables, and similarly variable updates are on the i-related entries; precisely, all

the entries at the indices $i +\!\!+ i''$ for some i'' are updated. The final score map of C is the function T' that maps each $i \in A$ to the total score computed by the execution under i.

If the command C is the sequential composition $(C_1; C_2)$, then C_1 is run first under given (D, σ, A) and C_2 is run afterwards under (D, σ_1', A) where σ_1' is the final state of C_1. The final state σ' of the entire C is the one from the execution of C_2, and the final score map T' of C is the *pointwise sum* of the maps T_1' and T_2' resulting from the executions of C_1 and C_2. If the command C is $(\textbf{for } x^{\mathsf{int}^\dagger} \textbf{ in range}(n_+) \textbf{ do } C)$, then C behaves the same as its loop-unrolled version.

If the command C is the conditional $(\textbf{ifz } Z\ C_1\ C_2)$, then its execution starts by *partitioning* the index set A into A_1 and A_2 such that A_1 contains all the indices $i \in A$ under which Z evaluates to zero, and A_2 contains the remaining indices. Then, C_1 is run under (D, σ, A_1), and C_2 is run under (D, σ_1', A_2) afterwards where σ_1' is the final state of C_1. The final state σ' of the conditional command C is the one from the execution of C_2, and the final score map T' of C is the following combination of T_1' and T_2' resulting from the runs of C_1 and C_2: $T'(i) = T_1'(i)$ for all $i \in A_1$ and $T'(i) = T_2'(i)$ for all $i \in A_2$. If the reader is familiar with abstract interpretation, she or he may have noticed that our conditional command behaves according to (a finite executable version of) the standard collecting semantics in abstract interpretation.

New commands. The remaining four cases are new. The first case enables a fine-grained control over *looping and score management*, and the other cases enable that over *vectorised parallel execution.*

If the command C is $(\textbf{loop\_fixpt\_noacc}(n_+)\ C_1)$, it behaves the same as the sequential composition of n_+-many copies of C_1's, except for two points. First, the execution of C skips some copies of C_1 and stops early if its *fixed-point check* gives a positive answer. Concretely, after the run of each copy of C_1, the execution checks whether the state has been unchanged during the run of this copy, that is, whether the state is a fixed point of the run of the copy. If the check says yes, the execution stops, skipping the remaining copies of C_1. This early stopping based on fixed-point check leads to the performance improvement in our optimisation. Second, the execution of C does not follow the *score accumulation scheme* used by the sequential composition, but it instead keeps the score of the last executed copy of C_1 only, ignoring the scores computed by all the other executed copies of C_1. This unusual treatment of scores is closely tied to the use of speculation in our optimisation; ignoring amounts to discarding the results of certain unconfirmed speculative computations.

If the command C is $(\textbf{extend\_index}(\alpha, n_+)\ C_1)$, it *extends* every index i in A by adding (α, k) at the end for all $k \in \{0, \ldots, n_+ - 1\}$; recall that C is assumed to be run under (D, σ, A). This extension may fail if $i +\!\!+ [(\alpha, k)]$ for some $i \in A$ and $k \in \{0, \ldots, n_+ - 1\}$ violates the definition of *Index* because the string α is already present in i. If this happens, the execution of C gets stuck. Otherwise, we have a new index set A' of size $|A| \times n$ consisting of the extended indices, and C_1 is then run under (D, σ, A'). When the execution of C_1 is completed, the original index set A is *restored*, and the final state σ_1' and score map T_1' of C_1 are *post-processed* to yield the final results, σ' and T', for C. Here T' is obtained from T_1' via summation: $T'(i) = \sum_{k=0}^{n_+ - 1} T_1'(i +\!\!+ [(\alpha, k)])$ for every $i \in A$. The state σ' is obtained by going through every variable x and updating the map $\sigma_1'(x)$ stored in x as follows: $\sigma'(x)(i') = \sigma_1'(x)(i +\!\!+ [(\alpha, n_+ - 1)])$ if $i' = i +\!\!+ i''$ for some $i \in A$ and i'', and $\sigma'(x)(i') = \sigma_1'(x)(i')$ otherwise. Intuitively, this post-processing step ensures that for all $i \in A$, σ' keeps only the result of the last $(i +\!\!+ [(\alpha, n_+ - 1)])$ execution of C_1, while T' keeps the scores computed by all the $(i +\!\!+ [(\alpha, \ell)])$ executions of C_1 for $\ell \in \{0, \ldots, n_+ - 1\}$.

If the command C is $(x^{\mathsf{int}^\dagger} := \textbf{lookup\_index}(\alpha))$, it first checks whether every index i in the set A includes a pair (α, ℓ_i) for some $\ell_i \in \mathbb{Z}$. If the answer is no, the execution of C gets stuck. Otherwise, for every $i \in A$, C stores ℓ_i at the $(i +\!\!+ i'')$ entry of the map of x, for all i''. This construct allows the i execution to depend on information *specific* to i, thereby enabling different components of vectorised execution to perform different computations.

$$
\begin{aligned}
\overline{x^{\mathsf{real}} := \mathbf{fetch}(I)} &\equiv x^{\mathsf{real}\dagger} := \mathbf{fetch}(\overline{I}), \\
\overline{\mathbf{score}(E)} &\equiv \mathbf{score}(\overline{E}), \\
\overline{x^{\mathsf{int}} := Z} &\equiv x^{\mathsf{int}\dagger} := \overline{Z}, \\
\overline{x^{\mathsf{real}} := E} &\equiv x^{\mathsf{real}\dagger} := \overline{E}, \\
\overline{\mathbf{skip}} &\equiv \mathbf{skip}, \\
\overline{C_1; C_2} &\equiv \overline{C_1}; \overline{C_2}, \\
\overline{\mathbf{ifz}\ Z\ C_1\ C_2} &\equiv \mathbf{ifz}\ \overline{Z}\ \overline{C_1}\ \overline{C_2},
\end{aligned}
$$

$$
\begin{aligned}
&\overline{\mathbf{for}\ x^{\mathsf{int}}\ \mathbf{in}\ \mathbf{range}(n)\ \mathbf{do}\ C_1} \\
&\quad\equiv \mathbf{extend\_index}(\alpha, n)\ \{ \\
&\qquad \mathbf{loop\_fixpt\_noacc}(n)\ \{ \\
&\qquad\quad \mathbf{shift}(\alpha); \\
&\qquad\quad x^{\mathsf{int}\dagger} := \mathbf{lookup\_index}(\alpha); \\
&\qquad\quad \overline{C_1}\ \}\}
\end{aligned}
$$

where α is a string not appearing in $\overline{C_1}$.

Fig. 5. Translation of commands in the target language.

If the command C is $\mathbf{shift}(\alpha)$, it updates all the maps stored in the variables by *moving* values inside each map in a manner analogous to the bitwise-shift operation. More precisely, assume that C is run under (D, σ, A) and $A = \{i +\!\!+ [(\alpha, k)] : i \in A_0,\ k \in \{0, \ldots, n_+ - 1\}\}$ for some $n_+ \geq 1$ and A_0 such that α is not already present in every index $i \in A_0$. Then, $\mathbf{shift}(\alpha)$ results in the state σ' by going through every variable x and updating the map $\sigma(x)$ stored in x as follows:

$$
\sigma'(x)(i') = \begin{cases} \sigma(x)(i +\!\!+ [(\alpha, k-1)]) & \text{if } i' = i +\!\!+ [(\alpha, k)] +\!\!+ i'' \text{ for some } k \in \{1, \ldots, n_+ - 1\} \text{ and } i'', \\ \sigma(x)(i) & \text{if } i' = i +\!\!+ [(\alpha, 0)] +\!\!+ i'' \text{ for some } i'', \\ \sigma(x)(i') & \text{otherwise.} \end{cases}
$$

A good way to understand this definition is to ignore the i'' part, and to focus on where the entries of σ' come from: the $(i +\!\!+ [(\alpha, k)])$ entry is obtained from the $(i +\!\!+ [(\alpha, k-1)])$ entry for all $k \in \{1, \ldots, n_+ - 1\}$, and the $(i +\!\!+ [(\alpha, 0)])$ entry is obtained from the i entry. This **shift** command lets different components of a vectorised parallel execution communicate with each other by moving values between them, and enables us to vectorise loops that have dependencies between iterations.

3.3 Translation

Our loop vectorisation is implemented as a translation from the source language to the target language. The translation assumes that these languages use the same set of symbols for variables, and for every variable x in that set, they give the corresponding types, i.e., x has a type τ in the source language if and only if it has the type $\tau\dagger$ in the target language. Using this assumption, the translation converts expressions of all three types in the source language to themselves except that variables x^{τ} in those expressions are replaced by $x^{\tau\dagger}$. We put a line on top of expressions in the source language to denote their translations. Thus, $\overline{E}$, $\overline{Z}$, and $\overline{I}$ mean the translated E, Z, and I.

The translation of commands is defined inductively by the rules in Fig. 5. The most important rule is the one for translating a for-loop and thereby implementing the main part of our loop vectorisation; the other rules simply apply the translation inductively.

Consider the translation of $C \equiv (\mathbf{for}\ x^{\mathsf{int}}\ \mathbf{in}\ \mathbf{range}(n)\ \mathbf{do}\ C_1)$ in the figure. The translated $\overline{C}$ is a **loop_fixpt_noacc** loop run under the scope of the $\mathbf{extend\_index}(\alpha, n)$ construct. A single iteration of this loop corresponds to the *vectorised parallel execution* of the body C_1 of the original for-loop over all the indices in $\mathbf{range}(n)$. The translation implements this parallel execution using the $\mathbf{extend\_index}(\alpha, n)$ construct, which changes the input index set A to $A' = \{i +\!\!+ [(\alpha, k)] : i \in A,\ k \in \{0, \ldots, n-1\}\}$ and increases the level of parallelism by the factor of n since $|A| \times n = |A'|$; the increased parallelism is then used to run all the n iterations of the loop body C_1 simultaneously.

$$C_2 = \left[\begin{array}{l} \textbf{extend\_index}(\text{“}\textbf{vec}\text{”}, 10)\ \{ \\ \quad \textbf{loop\_fixpt\_noacc}(10)\ \{ \\ \quad\quad \textbf{shift}(\text{“}\textbf{vec}\text{”});\ t^{\text{int}\dagger} := \textbf{lookup\_index}(\text{“}\textbf{vec}\text{”});\ C_0\ \}\}. \end{array}\right]$$

Fig. 6. Example translation of *HMM* in Fig. 1 to the target language. The command C_0 was defined in Fig. 3.

The parallel execution above is *speculative* in that the execution of the translated loop body $\overline{C_1}$ under an index $(i +\!\!+ [(\alpha, k{+}1)]) \in A'$ does not wait until the result of the previous iteration under the index $(i +\!\!+ [(\alpha, k)])$ is available—it just proceeds with a predicted result of the previous iteration.

To preserve the semantics of the original for-loop despite the speculative execution, the translation of C *repeats* the parallel execution up to n times using the **loop_fixpt_noacc**(n) construct, and forces *communication* between different threads of the parallel execution (which represent computations under different indices in A') using the **shift**(α) command. Each round of the repetition starts by running **shift**(α), which moves values between different threads such that the $(i +\!\!+ [(\alpha, k{+}1)])$-indexed thread for the $(k{+}1)$-th iteration of the loop body $\overline{C_1}$ in this round receives, at the start of execution, the result of the $(i +\!\!+ [(\alpha, k)])$-indexed thread for the k-th iteration from the previous round. Moving values this way gradually eliminates the speculative nature of the parallel execution: at the end of the j-th round, the threads for the $0, \ldots, j$-th iterations of $\overline{C_1}$ are guaranteed to start with correct results for the previous iterations so that they are no longer speculative.

Another important aspect of the translation of C is that the **loop_fixpt_noacc** loop is stopped early if the state is not changed by the previous round of the loop, which is detected by the *fixed-point check* of the **loop_fixpt_noacc** construct. This early stopping occurs often in practice, and it is crucial for the performance improvement of our loop vectorisation.

Finally, we point out that *only the last round* of the **loop_fixpt_noacc** loop contributes to the final score map T' computed by the loop; the score maps computed by all the other rounds of the loop are simply discarded. Using the score map of the last round only comes from the fact that only the parallel execution of the last round is guaranteed to be the non-speculative parallel execution of all the iterations of the original for-loop in the source language.

Example 3 (C_2 in Fig. 6)**.** Recall the *HMM* example from Fig. 1. Translating *HMM* to the target language gives the command C_2 in Fig. 6. In the rest of this subsection, we will go through the key steps of the execution of C_2, and explain how C_2 uses *speculative execution* and *fixed-point check* to vectorise the loop in *HMM* even though the loop iterations have data dependencies between them.

Let D be a random database as defined in Example 1. Also, let $A_{[]} = \{[]\}$ be the singleton set of the empty index, and σ_0 be the state where every variable stores the constant-zero function. Suppose that C_2 is run with $(D, \sigma_0, A_{[]})$. Then, the **loop_fixpt_noacc** loop inside C_2 performs two iterations, and C_2 produces the state σ' and the score map $T' : A_{[]} \to \mathbb{R}$ given by

$$\sigma'(t) = \lambda i.\, 9, \qquad \sigma'(x) = \sigma'(y) = \lambda i.\, z_9, \qquad \sigma'(v) = \sigma_0(v) \text{ for all other variables } v,$$
$$T'([]) = \textstyle\sum_{\ell=0}^{9} \Big(\log p_N(z_\ell; f(z_{\ell-1}), 1.0) + \log p_N(o(\ell); g(z_\ell), 1.0) \Big),$$

where $z_{-1} = 0.0$. At the index [], this outcome matches that of *HMM* in the source language. Specifically, $T'([])$ equals to r_{HMM} from Eq. (5), and for every variable u, the value $\sigma'(u)([])$ equals to the value of u in the final state of *HMM*. We now examine the execution of C_2 in detail.

First round of the loop. The execution of C_2 begins by extending indices in $A_{[]}$ with (“**vec**”, ℓ) for all $\ell \in \{0, \ldots, 9\}$, and updating the current index set to that of the extended indices, i.e., $A_{vec} = \{i_0, \ldots, i_9\}$ for $i_\ell = [(\text{“}\textbf{vec}\text{”}, \ell)]$. Then, it runs the body of the loop of C_2 with (D, σ_0, A_{vec}). The first command in the loop body is **shift**(“**vec**”), and it acts as no-op (or **skip**) this time, because all the variables in σ_0 store constant-zero maps and so moving values between entries does not

change those maps. The next command looks up the value bound to "**vec**" in each index i in A_{vec}, stores the looked-up value at the i-related entries of the map of t (i.e., entries at indices $i +\!\!+ i'$ for all i'), and returns the following state $\hat{\sigma}_0$:

$$\hat{\sigma}_0 = \sigma_0[t \mapsto m_t] \;\text{ where } m_t(i) = \text{if } \left(i = i_\ell +\!\!+ i' \text{ for some } \ell \in \{0,\ldots,9\} \text{ and } i'\right) \text{ then } \ell \text{ else } \sigma_0(t)(i).$$

The rest of the loop body executes equivalently to C_0 in Example 1, producing the same outcome: a state σ'_0 and a score map $T'_0 : A_{vec} \to \mathbb{R}$ given by

$$\sigma'_0 = \sigma_0[t \mapsto m_t, x \mapsto m'_0, y \mapsto m'_0], \quad m'_0(i) = \begin{cases} z_\ell & \text{if } i = i_\ell +\!\!+ i' \text{ for some } \ell \in \{0,\ldots,9\} \text{ and } i', \\ 0.0 & \text{otherwise,} \end{cases}$$

$$T'_0(i_\ell) = \log p_{\mathcal{N}}(z_\ell; f(0.0), 1.0) + \log p_{\mathcal{N}}(o(\ell); g(z_\ell), 1.0) \quad \text{for all } \ell \in \{0,\ldots,9\}.$$

Note that for all $i_\ell \in A_{vec}$, the i_ℓ part of the execution of the loop body corresponds to the ℓ-th iteration of the original for-loop of *HMM*, but this correspondence is not perfect due to the *incorrect speculation* used by the former: the i_ℓ part of the execution speculates that in the original for-loop of *HMM*, the variable y stores $\sigma_0(y)(i_\ell) = 0.0$ after the $(\ell-1)$-th iteration, but if $\ell \geq 1$ and $z_{\ell-1} \neq 0$, the speculation is wrong because the correct value of y is $z_{\ell-1}$. As a result of incorrect speculation and imperfect correspondence, $T'(i_\ell)$ is not equal to the score of the ℓ-th iteration of *HMM* in general; the former uses $f(0.0)$ instead of $f(z_{\ell-1})$.

The incorrect speculation is detected by the *fixed-point check* of the **loop_fixpt_noacc** construct of C_2, and it is corrected also by the *re-execution* of the body of the **loop_fixpt_noacc** loop with improved speculation. Concretely, the fixed-point check of **loop_fixpt_noacc** tests whether the state σ'_0 is the same as the initial state σ_0, and gets a negative answer, which indicates that the speculated value of y at some iteration of the for-loop of *HMM* is wrong. Because of the negative answer, the body of **loop_fixpt_noacc** is executed again but this time speculation is made based on the current state σ'_0, instead of the old state σ_0. Using the more recent state for speculation always leads to improvement in terms of correctness.

Subsequent rounds. In our example, the speculation made at the j-th round of **loop_fixpt_noacc** is guaranteed to be correct for the k-th iteration of the original for-loop of *HMM* for all $k \leq j$ where correctness means that the speculated value of y at the beginning of the k-th iteration equals the actual value of y at that iteration of the for-loop of *HMM*. In the example, we have more than this guaranteed improvement, because all the speculated values of y at the second round of the **loop_fixpt_noacc** loop are correct. The use of the *correct speculation* implies that the state σ'_1 after the second round equals σ'_0, and so the fixed-point check of **loop_fixpt_noacc** succeeds this time and the **loop_fixpt_noacc** loop terminates after this second round.

The final state σ' and the score map T' of C_2 are then constructed from σ'_1 and the score map T'_1 computed by the second round, according to the semantics of **extend_index**("**vec**", 10):

$$\sigma' = \sigma'_0[t \mapsto m'_t,\ x \mapsto m',\ y \mapsto m'], \qquad m'_t = \lambda i.\, 9, \qquad m' = \lambda i.\, z_9,$$
$$T' = \left[[] \mapsto \textstyle\sum_{\ell=0}^{9} \left(\log p_{\mathcal{N}}(z_\ell; f(z_{\ell-1}), 1.0) + \log p_{\mathcal{N}}(o(\ell); g(z_\ell), 1.0) \right) \right].$$

Note that the final σ' and T' of C_2 at the index $[]$ are the same as the results of *HMM* in Fig. 1. □

4 Semantics of the Target Language and Soundness of Translation

In this section, we present an operational semantics of the target language, formalising the intuitive description of the language in §3.1 and §3.2 (§4.1). We then prove that the translation from the source language to the target language, described in §3.3, preserves the semantics (§4.2).

4.1 Semantics of the Target Language

Definition of states. The semantics relies on the standard prefix order $\sqsubseteq$ on indices: for all $i, j \in \mathit{Index}$, we have $i \sqsubseteq j$ if and only if $|i| \leq |j|$ and $i.k = j.k$ for all $k \in \{1, \ldots, |i|\}$. Here $|i|$ means the length of i, and $i.k$ means the k-th element of i. We denote the upward and downward closures of an index i and an index set L using the following notations: $i\uparrow = \{j \in \mathit{Index} : i \sqsubseteq j\}$, $i\downarrow = \{j \in \mathit{Index} : j \sqsubseteq i\}$, $L\uparrow = \bigcup_{j \in L} j\uparrow$, and $L\downarrow = \bigcup_{j \in L} j\downarrow$. Also, we let

$$\max(L) = \text{if } \big(i \text{ is the } \sqsubseteq\text{-largest element of } L\big) \text{ then } i \text{ else undefined.}$$

Thus, $i = \max(L)$ if and only if $i \in L$ and for all $i' \in L$, $i' \sqsubseteq i$. Note that if $L \cap j\downarrow \neq \emptyset$ for some index j, then $\max(L \cap j\downarrow)$ always exists, because $L \cap j\downarrow$ is totally ordered by $\sqsubseteq$.

For each type $\tau \in \{\mathsf{int}, \mathsf{real}\}$, let $[\![\mathsf{int}]\!] = \mathbb{Z}$ and $[\![\mathsf{real}]\!] = \mathbb{R}$. A **state** σ in the target language is a function sending variables to finite partial maps from indices to integers or reals such that for every variable $x^{\tau\dagger}$, (i) the range of the partial map $m_x = \sigma(x)$ is $[\![\tau]\!]$, and (ii) the domain of m_x contains the empty index $[]$. A recommended reading is to view the partial map m_x here as a data structure representing the following total function $\mathit{extend}(m_x)$:

$$\mathit{extend}(m_x) : \mathit{Index} \to [\![\tau]\!], \quad \mathit{extend}(m_x)(i) = m_x\big(\max(\mathsf{dom}(m_x) \cap i\downarrow)\big) \text{ for all } i \in \mathit{Index}. \tag{6}$$

Note that for every $i \in \mathit{Index}$, the set $(\mathsf{dom}(m_x) \cap i\downarrow)$ is nonempty because it contains the empty index $[]$ at least, and the set is also totally ordered because $i\downarrow$ is totally ordered. As a result, for all i, $\max(\mathsf{dom}(m_x) \cap i\downarrow)$ is a well-defined element in $\mathsf{dom}(m_x)$, which in turn implies that $\mathit{extend}(m_x)$ is a well-defined total map. Although the semantics of the target language to be presented shortly is defined in terms of finite partial maps stored in variables in states, it depends only on the total functions represented by those partial maps: for all states σ, σ', if $\mathit{extend}(\sigma(x)) = \mathit{extend}(\sigma'(x))$ for all variables x, the semantics is unable to distinguish between σ and σ'. When we explain the semantics in the rest of the paper, we will often refer to the total function $\mathit{extend}(\sigma(x))$ as the function stored in a variable x, although $\mathit{extend}(\sigma(x))$ is not stored in x. We write State_{tgt} for the set of all the states in the target language.

Our *extend* operation models *tensor broadcasting* [68] as implemented in popular tensor libraries such as PyTorch. To see this, first note that the operation can be applied to any finite partial map $m : \mathit{Index} \rightharpoonup V$ for a set V, even when the empty index $[]$ is not in $\mathsf{dom}(m)$. The result $\mathit{extend}(m)$ is still well-defined in this general case, although it might not be a total map. Next, as an example, consider two integer-type tensors t_1 and t_2 with shapes (3) and $(2, 3)$, respectively. In our setup, they are represented as partial maps $m_1, m_2 : \mathit{Index} \rightharpoonup \mathbb{Z}$ with $\mathsf{dom}(m_1) = \{[(\mathbf{snd}, \ell_2)] : \ell_2 \in \{0, 1, 2\}\}$ and $\mathsf{dom}(m_2) = \{[(\mathbf{snd}, \ell_2); (\mathbf{fst}, \ell_1)] : \ell_2 \in \{0, 1, 2\}, \ell_1 \in \{0, 1\}\}$. Although t_1 and t_2 have different shapes, they can be added using tensor broadcasting: the addition first broadcasts t_1 into a tensor of shape $(2, 3)$, and then performs pointwise addition with t_2. In our setup, this broadcasting is modelled by applying *extend* to m_1, which yields a partial map defined at all indices in $\mathsf{dom}(m_2)$, with each value corresponding to the broadcasted entries of t_1. The pointwise addition is then modelled by computing the elementwise sum of $\mathit{extend}(m_1)$ and m_2 at all indices in $\mathsf{dom}(m_2)$. Lastly, note that $\mathit{extend}(m_1)$ models not only this specific instance of broadcasting t_1 with respect to t_2, but also all possible instances of broadcasting t_1 with respect to any tensor t_2' of a larger "compatible" shape.

Semantics of target language. The semantics uses the following five semantic domains and their structures: *Index*, State_{tgt}, *FAChain*, $\mathit{Tensor}_{\mathbb{Z}}$, and $\mathit{Tensor}_{\mathbb{R}}$. We have already defined the first two in this list. The next is *FAChain*, the domain for finite antichains of indices. A set of indices L is an **antichain** if for all $i, j \in L$ with $i \neq j$, we have neither $i \sqsubseteq j$ nor $j \sqsubseteq i$. The domain *FAChain* consists of all the **finite antichains** of indices. When denoting an element of *FAChain*, we often use the symbol A, instead of the usual symbol L for an index set, in order to emphasise that it is

$$[\![n]\!](\sigma, i) = n, \qquad [\![x^{\mathsf{int}\dagger}]\!](\sigma, i) = \mathit{extend}(\sigma(x^{\mathsf{int}\dagger}))(i),$$
$$[\![\mathrm{op}_i(Z_1, \ldots, Z_k, E_1, \ldots, E_m)]\!](\sigma, i) = [\![\mathrm{op}_i]\!]_a([\![Z_1]\!](\sigma, i), \ldots, [\![Z_k]\!](\sigma, i), [\![E_1]\!](\sigma, i), \ldots, [\![E_m]\!](\sigma, i)),$$
$$[\![r]\!](\sigma, i) = r, \qquad [\![x^{\mathsf{real}\dagger}]\!](\sigma, i) = \mathit{extend}(\sigma(x^{\mathsf{real}\dagger}))(i),$$
$$[\![\mathrm{op}_r(Z_1, \ldots, Z_k, E_1, \ldots, E_m)]\!](\sigma, i) = [\![\mathrm{op}_r]\!]_a([\![Z_1]\!](\sigma, i), \ldots, [\![Z_k]\!](\sigma, i), [\![E_1]\!](\sigma, i), \ldots, [\![E_m]\!](\sigma, i)),$$
$$[\![[(\alpha_1, Z_1); \ldots; (\alpha_k, Z_k)]]\!](\sigma, i) = [(\alpha_1, [\![Z_1]\!](\sigma, i)); \ldots; (\alpha_k, [\![Z_k]\!](\sigma, i))].$$

Fig. 7. Semantics of expressions in the target language.

not just any index set, but an antichain with a finite number of elements. The last two domains are instances of the set of all V-**tensors** for a set V, which are finite partial maps from *Index* to V. This V-tensor set is written as Tensor_V. We use the symbol T to denote an element of Tensor_V.

The interpretations of expressions in the target language have the following forms: $[\![Z]\!] : \mathit{State}_{tgt} \times \mathit{Index} \to \mathbb{Z}$, $[\![E]\!] : \mathit{State}_{tgt} \times \mathit{Index} \to \mathbb{R}$, and $[\![I]\!] : \mathit{State}_{tgt} \times \mathit{Index} \to \mathit{Index}$. The defining clauses for these interpretations appear in Fig. 7. They assume that the interpretations of the integer-returning atomic operations op_i and the real-returning atomic operations op_r are given by $[\![\mathrm{op}_i]\!]_a$ and $[\![\mathrm{op}_r]\!]_a$, respectively. The clauses say that an expression is evaluated under a state σ and an index i in a standard way using the assumed $[\![\mathrm{op}_i]\!]_a$ and $[\![\mathrm{op}_r]\!]_a$ except that all the variable reads in the expression access the i entities of the represented functions for those variables.

The semantics of commands in the target language is specified in terms of a big-step computation relation $\Downarrow_{tgt} \subseteq (\mathit{Com}_{tgt} \times \mathit{RDB} \times \mathit{State}_{tgt} \times \mathit{FAChain}) \times (\mathit{State}_{tgt} \times \mathit{Tensor}_{\mathbb{R}})$. The relation $\Downarrow_{tgt}$ works on states in State_{tgt} and takes, as its inputs, a command C to run, a random database D, a state σ, and a finite antichain A of indices. Intuitively, the indices in A correspond to the ids of the threads that run the same C in parallel. The id of a thread determines, for each variable, which entries of the map stored in the variable are read and updated during the execution of C by the thread.

Operations on states. The rules for the relation $\Downarrow_{tgt}$ use the below notations on tensors and states. For a set of indices L, we write T^z_L for the tensor in $\mathit{Tensor}_{\mathbb{R}}$ that maps every index in L to 0.0 and is undefined for all other indices. Also, we write $\oplus$ for the following pointwise addition on $\mathit{Tensor}_{\mathbb{R}}$:

$$\oplus : \mathit{Tensor}_{\mathbb{R}} \times \mathit{Tensor}_{\mathbb{R}} \to \mathit{Tensor}_{\mathbb{R}}, \quad (T \oplus T')(i) = \begin{cases} T(i) + T'(i) & \text{if } i \in \mathrm{dom}(T) \cap \mathrm{dom}(T'), \\ T(i) & \text{if } i \in \mathrm{dom}(T) \setminus \mathrm{dom}(T'), \\ T'(i) & \text{if } i \in \mathrm{dom}(T') \setminus \mathrm{dom}(T), \\ \text{undefined} & \text{otherwise.} \end{cases}$$

Regarding states, the clauses in the figure use the update operation $\sigma[x^{\tau\dagger} : T]$ for a variable $x^{\tau\dagger}$ and a tensor $T \in \mathit{Tensor}_{[\![\tau]\!]}$, and the copy operation $\sigma\langle\rho\rangle$ for a finite partial injective map ρ on indices. These operations result in states $\sigma[x^{\tau\dagger} : T]$ and $\sigma\langle\rho\rangle$ in State_{tgt} defined as follows:

$$\sigma[x^{\tau\dagger} : T](y^{\tau'\dagger})(i) = \begin{cases} T(i) & \text{if } y^{\tau'\dagger} \equiv x^{\tau\dagger} \text{ and } i \in \mathrm{dom}(T), \\ \text{undefined} & \text{if } y^{\tau'\dagger} \equiv x^{\tau\dagger} \text{ and } i \in \mathrm{dom}(T)\uparrow \setminus \mathrm{dom}(T), \\ \sigma(y^{\tau'\dagger})(i) & \text{otherwise,} \end{cases}$$

$$\sigma\langle\rho\rangle(y^{\tau'\dagger})(i) = \begin{cases} \sigma(y^{\tau'\dagger})(\rho^{-1}(i)) & \text{if } i \in \mathrm{image}(\rho), \\ \text{undefined} & \text{if } i \in \mathrm{image}(\rho)\uparrow \setminus \mathrm{image}(\rho), \\ \sigma(y^{\tau'\dagger})(i) & \text{otherwise.} \end{cases}$$

If a partial map on the right-hand side is applied to an index that is not in its domain, the result is undefined, and we do not write this explicitly in the equations. Fig. 8 shows how update and copy operations work for simple partial maps.

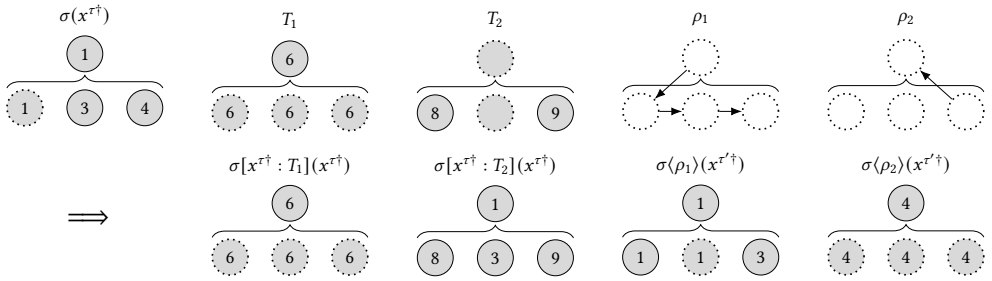

Fig. 8. Examples for update and copy. Nodes over braces represent the empty indices, and nodes under braces represent indices $i_\ell = [(\text{“}\mathbf{rv}\text{”}, \ell)]$ for $\ell \in \{0, 1, 2\}$. Values in solid nodes are stored in partial maps, while values in dotted nodes are induced by *extend*. Thus, $\sigma(x^{\tau\dagger}) = [[] \mapsto 1, i_1 \mapsto 3, i_2 \mapsto 4]$, $T_1 = [[] \mapsto 6]$, and $T_2 = [i_0 \mapsto 8, i_2 \mapsto 9]$. Also, $\mathit{extend}(\sigma(x^{\tau\dagger}))(i_0) = 1$, and $\mathit{extend}(T_1)(i_\ell) = 6$ for $\ell \in \{0, 1, 2\}$. The functions ρ_1 and ρ_2 denote the partial injections on *Index* used in the semantics for **shift** and **extend_index** (Fig. 9), respectively, and are depicted graphically in the top right; e.g., ρ_2 is defined only on i_2 and maps it to $[]$.

A good way to understand these slightly unusual update and copy operations on states is to recall that finite partial maps stored in variables represent total functions obtained by the *extend* operation, and examine what the operations do on these represented functions. For a function $f : \mathit{Index} \to V$ and a partial function $g : \mathit{Index} \rightharpoonup V$, let $f[g]$ be the outcome of updating f with g, that is, the function mapping each index $i \in \mathrm{dom}(g)$ to $g(i)$ and every other index i to its original value $f(i)$. The next lemma characterises the effects of our update and copy operations by means of standard update and copy operations on the represented functions. Its proof is in §A.2.

Lemma 1. *Let σ be a state, $x^{\tau\dagger}$, $y^{\tau'\dagger}$ be variables, T be a tensor in $\mathit{Tensor}_{[\![\tau]\!]}$, and ρ be a finite partial injection on Index. Then, we have the following equations:*

$$\mathit{extend}(\sigma[x^{\tau\dagger} : T](y^{\tau'\dagger})) = \text{if } \left(y^{\tau'\dagger} \equiv x^{\tau\dagger}\right) \text{ then } \mathit{extend}(\sigma(x^{\tau\dagger}))[\mathit{extend}(T)] \text{ else } \mathit{extend}(\sigma(y^{\tau'\dagger})),$$
$$\mathit{extend}(\sigma\langle\rho\rangle(y^{\tau'\dagger})) = \mathit{extend}(\sigma(y^{\tau'\dagger}))[\mathit{extend}(T_{y,\rho})],$$

where $T_{y,\rho}(i) = \sigma(y^{\tau'\dagger})(\rho^{-1}(i))$ if $i \in \rho(\mathrm{dom}(\sigma(y^{\tau'\dagger})))$, and it is undefined otherwise.

Here we apply the *extend* operation in Eq. (6) to the tensor T, although T may be undefined at the empty index unlike in the case of the partial maps stored in states. As a result, this application may lead to a partial function of type $\mathit{Index} \rightharpoonup [\![\tau]\!]$, which we recall below:

$$\mathit{extend}(T)(i) = \text{if } \left(i \in \mathrm{dom}(T)\uparrow\right) \text{ then } T(\max(\mathrm{dom}(T) \cap i\downarrow)) \text{ else undefined.}$$

Note that the equations in the lemma use $(\_)[\mathit{extend}(T)]$ and $(\_)[\mathit{extend}(T_{y,\rho})]$, instead of $(\_)[T]$ and $(\_)[T_{y,\rho}]$. This means that both operations change more entries than one typically expects. For instance, in the first equation, each $(i, T(i))$ entry of the partial map T results in the changes of all the entries in $\mathit{extend}(\sigma(x^{\tau\dagger}))$ whose indices are in $i\uparrow$. We use this aggressive version of change in our operations because it enables us to implement the operations and our target language easily and efficiently using the tensor-broadcasting mechanism of PyTorch.

This aggressive change also helps maintain the following important invariant of our target language. For an index set L, we say a state σ is an L-**state** if $\mathrm{dom}(\sigma(x^{\tau\dagger})) \subseteq L\downarrow$ for all variables $x^{\tau\dagger}$. Intuitively, it means that for every variable $x^{\tau\dagger}$, if the partial map $\sigma(x^{\tau\dagger})$ stored in $x^{\tau\dagger}$ represents a function $f = \mathit{extend}(\sigma(x^{\tau\dagger}))$, then the function is determined by what it does on $L\downarrow$, i.e., $f = \mathit{extend}(f|_{L\downarrow})$. The invariant is that if a command C is executed under a finite antichain A and an input state is an A-state, then the output state is also an A-state. This invariant supports the view that the command C is run in parallel by the threads with their ids in A and the state consists of the disjoint parts for these threads. Formal lemmas and proofs are detailed in §A.4.

$$\frac{T = [i \mapsto D([\![I]\!](\sigma, i)) : i \in A]}{(x^{\text{real}\dagger} := \textbf{fetch}(I)), D, \sigma, A \Downarrow_{tgt} \sigma[x^{\text{real}\dagger} : T], T_A^z} \qquad \frac{T = [i \mapsto [\![E]\!](\sigma, i) : i \in A]}{\textbf{score}(E), D, \sigma, A \Downarrow_{tgt} \sigma, T}$$

$$\frac{T = [i \mapsto [\![Z]\!](\sigma, i) : i \in A]}{(x^{\text{int}\dagger} := Z), D, \sigma, A \Downarrow_{tgt} \sigma[x^{\text{int}\dagger} : T], T_A^z} \qquad \frac{T = [i \mapsto [\![E]\!](\sigma, i) : i \in A]}{(x^{\text{real}\dagger} := E), D, \sigma, A \Downarrow_{tgt} \sigma[x^{\text{real}\dagger} : T], T_A^z}$$

$$\frac{}{\textbf{skip}, D, \sigma, A \Downarrow_{tgt} \sigma, T_A^z} \qquad \frac{C_1, D, \sigma, A \Downarrow_{tgt} \sigma', T' \quad C_2, D, \sigma', A \Downarrow_{tgt} \sigma'', T''}{(C_1; C_2), D, \sigma, A \Downarrow_{tgt} \sigma'', (T' \oplus T'')}$$

$$\frac{\begin{array}{c} A_1 = \{i \in A : [\![Z]\!](\sigma, i) = 0\} \quad C_1, D, \sigma, A_1 \Downarrow_{tgt} \sigma', T' \\ A_2 = \{i \in A : [\![Z]\!](\sigma, i) \neq 0\} \quad C_2, D, \sigma', A_2 \Downarrow_{tgt} \sigma'', T'' \end{array}}{(\textbf{ifz}\ Z\ C_1\ C_2), D, \sigma, A \Downarrow_{tgt} \sigma'', (T' \oplus T'')}$$

$$\frac{\sigma_0 = \sigma \qquad C, D, \sigma_k[x^{\text{int}\dagger} : [i \mapsto k \ : \ i \in A]], A \Downarrow_{tgt} \sigma_{k+1}, T_{k+1} \text{ for all } k \in \{0, \ldots, n_+ - 1\}}{(\textbf{for}\ x^{\text{int}\dagger}\ \textbf{in range}(n_+)\ \textbf{do}\ C), D, \sigma, A \Downarrow_{tgt} \sigma_{n_+}, \bigoplus_{k=1}^{n_+} T_k}$$

$$\frac{\alpha \in \text{dom}(i) \text{ for all } i \in A \qquad T = [i \mapsto i(\alpha) \ : \ i \in A]}{(x^{\text{int}\dagger} := \textbf{lookup\_index}(\alpha)), D, \sigma, A \Downarrow_{tgt} \sigma[x^{\text{int}\dagger} : T], T_A^z}$$

$$\frac{\begin{array}{c} \alpha \notin \text{dom}(i) \text{ for all } i \in A \quad A' = \{i +\!\!+ [(\alpha, k)] : i \in A,\ k \in \{0, \ldots, n_+ - 1\}\} \quad C, D, \sigma, A' \Downarrow_{tgt} \sigma', T' \\ \rho = [i +\!\!+ [(\alpha, n_+ - 1)] \mapsto i : i \in A] \qquad T'' = [i \mapsto \textstyle\sum_{0 \leq k < n_+} T'(i +\!\!+ [(\alpha, k)]) : i \in A] \end{array}}{(\textbf{extend\_index}(\alpha, n_+)\ C), D, \sigma, A \Downarrow_{tgt} \sigma' \langle \rho \rangle, T''}$$

$$\frac{\begin{array}{c} 0 \leq m < n_+ - 1 \qquad \sigma_0 = \sigma \\ C, D, \sigma_k, A \Downarrow_{tgt} \sigma_{k+1}, T_{k+1} \text{ and } \sigma_k \neq \sigma_{k+1} \text{ for all } k \in \{0, \ldots, m-1\} \qquad C, D, \sigma_m, A \Downarrow_{tgt} \sigma_m, T_{m+1} \end{array}}{(\textbf{loop\_fixpt\_noacc}(n_+)\ C), D, \sigma, A \Downarrow_{tgt} \sigma_m, T_{m+1}}$$

$$\frac{\sigma_0 = \sigma \quad C, D, \sigma_k, A \Downarrow_{tgt} \sigma_{k+1}, T_{k+1} \text{ for all } k \in \{0, \ldots, n_+ - 1\} \quad \sigma_k \neq \sigma_{k+1} \text{ for all } k \in \{0, \ldots, n_+ - 2\}}{(\textbf{loop\_fixpt\_noacc}(n_+)\ C), D, \sigma, A \Downarrow_{tgt} \sigma_{n_+}, T_{n_+}}$$

$$\frac{\rho(i) = \begin{cases} i +\!\!+ [(\alpha, 0)] & \text{if } \alpha \notin \text{dom}(i) \wedge i +\!\!+ [(\alpha, 0)] \in A, \\ j +\!\!+ [(\alpha, m+1)] & \text{if } i = j +\!\!+ [(\alpha, m)] \in A{\downarrow} \wedge j +\!\!+ [(\alpha, m+1)] \in A \text{ for some } j \text{ and } m \geq 0, \\ \text{undefined} & \text{otherwise} \end{cases}}{\textbf{shift}(\alpha), D, \sigma, A \Downarrow_{tgt} \sigma \langle \rho \rangle, T_A^z}$$

Fig. 9. Semantics of commands in the target language.

Semantics of commands. The rules for the $\Downarrow_{tgt}$ relation in Fig. 9 mostly follow the general principle of running each command over multiple input data specified by the input antichain A, except for two unusual parts. First, updating $x^{\tau\dagger}$ with operation $\sigma[x^{\tau\dagger} : T]$ modifies the partial map $\sigma(x^{\tau\dagger})$ beyond typical SIMD behaviour: if an entry in $\sigma(x^{\tau\dagger})$ has the index $i \in \text{dom}(T)\uparrow$, the entry gets updated (if $i \in \text{dom}(T)$) or removed (if $i \in \text{dom}(T)\uparrow \setminus \text{dom}(T)$). Second, the rule for the conditional command in the semantics is unusual. It splits the input index set A into A_1 and A_2, where A_1 consists of indices leading to the true branch and A_2 those leading to the false branch. The command then runs the true branch under A_1, and then the false branch under A_2. The disjointedness of A_1 and A_2, and the fact that $A_1 \cup A_2$ is an antichain prevent interference between the two branches: they do not write to the same entry of a variable, and neither of these branches reads from an entry of a variable written by the other branch. This non-interference means the execution order of the true and false branches is interchangeable without changing the semantics of the command.

We now go through the last five rules of the $\Downarrow_{tgt}$ relation in Fig. 9, which describe the behaviour of the new constructs in the target language. The rule for the **lookup_index** command says that the command regards each index $i = [(\alpha_1, m_1); \ldots; (\alpha_k, m_k)] \in A$ as a partial function mapping string α_i to integer m_i, and it runs only when α is in the domain of this partial function (i.e., $\alpha \in \mathrm{dom}(i)$). In that case, the command looks up the α entry of the partial function i for all $i \in A$, and stores the results of these lookups in the variable $x^{\mathsf{int}\dagger}$. The next rule is concerned with the **extend_index** command, and says that the command works in two steps. First, it runs the command C in its body under the set A' of extended indices, where A' consists of the indices $i +\!\!+ [(\alpha, k)]$ for all $i \in A$ and $k \in \{0, \ldots, n-1\}$. Second, it goes through every variable $x^{\tau\dagger}$, accesses the partial map m_x stored in $x^{\tau\dagger}$, and copies, for every $i \in A$, the value $m_x(i +\!\!+ [(\alpha, n-1)])$ to the i entry of m_x while removing the entries of m_x whose indices are in $i\!\uparrow \setminus \{i\}$. So, the copy here clears the entries at indices $i +\!\!+ [(\alpha, k)]$ for $k \in \{0, \ldots, n-1\}$ if such entries exist in m_x. Note that the copy is implemented with the help of the $(\_)\langle\rho\rangle$ operation (Fig. 8). Also, note that the tensor T'' in the result of the **extend_index** command is constructed from the values of the $i +\!\!+ [(\alpha, k)]$ entries in T' for all $i \in A$ and $k \in \{0, \ldots, n-1\}$. If some of these entries are not defined, the execution does not proceed. But this failure case does not occur, as indicated by Lemma 6 in §A, because the domain of the tensor in the result of execution is always the same as the given input antichain for the execution, and so $\mathrm{dom}(T') = A'$.

The following two rules describe the behaviour of the (**loop_fixpt_noacc**(n_+) C) command. They say that the command repeatedly runs its body C up to n_+ times, but unlike the usual loop, this repetition may stop early at some iteration $m{+}1$, and skip the remaining $n_+{-}(m{+}1)$ iterations; this early stopping happens when a fixed point is reached at the iteration m (for the first time) and so the fixed-point check at the following iteration succeeds. Another important point is that the (**loop_fixpt_noacc**(n_+) C) command does not accumulate the score maps computed during iterations, and keeps only the score map at the last iteration (which is T_{m+1} in the case of early stopping and T_{n+} otherwise). Discarding the score maps from all the non-last iterations comes from our loop vectorisation; it is exploited by our translation for vectorisation. The last rule describes the behaviour of the **shift** command: for every variable $x^{\tau\dagger}$ in the current state, it looks up the partial map m_x stored in the variable $x^{\tau\dagger}$, and shifts values in m_x according to ρ in the rule, using the $(\_)\langle\rho\rangle$ operation (Fig. 8). When $\rho(i) = i'$, the definition of ρ ensures that $i \in A\!\downarrow$ and $i' \in A$ for the input antichain A so that the source index of the shift is always in $A\!\downarrow$ and the target index is in A.

4.2 Soundness of Translation

The next theorem states that our parallelising translation preserves the semantics.

Theorem 1. *Let C be a command in the source language, and $\overline{C}$ be the parallelising translation of C in the target language. Let $D \in RDB$, $\sigma_0 \in State_{src}$, and $\sigma_1 \in State_{tgt}$. If $\sigma_1(x^{\tau\dagger}) = [[] \mapsto \sigma_0(x^\tau)]$ for all variables x^τ in the source language, there exist $\sigma'_0 \in State_{src}$, $\sigma'_1 \in State_{tgt}$, and $r \in \mathbb{R}$ such that*

$$C, D, \sigma_0 \Downarrow \sigma'_0, r, \quad \overline{C}, D, \sigma_1, \{[]\} \Downarrow_{tgt} \sigma'_1, [[] \mapsto r], \quad \text{and} \quad \sigma'_1(x^{\tau\dagger}) = [[] \mapsto \sigma'_0(x^\tau)] \ \text{ for all } x^\tau.$$

Intuitively, Theorem 1 says that if a command C in the source language and its translation $\overline{C}$ in the target language are run in the same states, then C and $\overline{C}$ terminate successfully, and their final states and final scores are essentially the same. The proof of Theorem 1 is provided in §A, where the theorem is restated as Corollary 5.

The key point of the proof stems from the fact that $\overline{C}$ may store in a single target language state information about multiple source language states. Specifically, when C contains a for-loop, $\overline{C}$ executes all the iterations of this loop in parallel. Thus, each of the states that arise during the execution of the translated loop in $\overline{C}$ corresponds to multiple states that arise during the execution of the original loop in C. The difficulty is to capture this many-to-one correspondence between

states in the source and target languages, and to prove its invariance so as to establish the theorem. We address the challenge by first embedding states and commands in the source language to those in the target language, then defining a form of simulation relation between states of the target language, which we call a **coupling** relation, and finally proving that modulo embedding, the relation is preserved by the execution of a command in the source language and that of its translation in the target language. For the details of the proof, see §A.

5 Implementation

We implemented our vectorisation technique for Pyro [9], a probabilistic programming language built on PyTorch [50]. Our choice of Pyro is motivated by its powerful support for parallel execution, which leverages PyTorch's efficient tensor operations, tensor broadcasting, and automatic differentiation, and by its rich set of inference engines, including Stochastic Variational Inference (SVI) and variable elimination. We point out that our implementation is not specific to Pyro and can be adapted to other PPLs that support similar features. Beyond standard Pyro/PyTorch primitives, our design incorporates a non-standard runtime to handle variable reading and writing. This runtime reflects the SIMD-style semantics of expressions and assignment commands based on the antichains in our target language. We also define two novel primitives specifically for the target language's control flow (**ifz**) and loop (**for**) constructs. These functionalities are integrated into a library that overrides standard Python variable access and provides new constructors. The key aspects of our implementation are summarised below.

Representing finite partial maps with PyTorch tensors. Our runtime relies on PyTorch tensors to represent finite partial maps from *Index* in our target language, such as V-tensors T and partial maps m stored in variables. This approach, however, needs to address several representation problems. One of these is that PyTorch uses integer sequences as indices, but our target language uses string-integer sequences as indices. To bridge this gap, our library maintains a global partial map from strings to integers, allocating a unique dimension for each string, as shown in Fig. 10. The other problem is that a finite partial map in our target language accepts indices of varying length, while a PyTorch tensor assumes indices of a fixed length. To address this problem, we track the maximum integer associated with each string present in the partial map's domain. We then construct a PyTorch tensor where the maximum index of each tensor dimension is one larger than the tracked maximum integer of the corresponding string. For instance, in Fig. 10, a partial map T has strings α and β with the maximum integers 9 and 8, respectively. Therefore, it is represented by a PyTorch tensor of shape $(11, 10)$, where the first dimension corresponds to α and the second to β by the global mapping. Here, the one additional component in each dimension of the PyTorch tensor, accessed by -1 in the figure, is used to represent the case that the corresponding string is absent in an index of the partial map. Each cell of the PyTorch tensor is then assigned the value evaluated by the $\mathit{extend}(T)$ function at the corresponding index. For example, in Fig. 10, for all $0 \leq i \leq 9$ and $0 \leq j \leq 8$, the cell of the PyTorch tensor at index $[i, j]$ is assigned the value of $\mathit{extend}(T)([(\alpha, i); (\beta, j)]) = c_{i,j}$, the cell at index $[i, -1]$ is set to the value of $\mathit{extend}(T)([(\alpha, i)]) = b_i$, the cell at index $[-1, j]$ is given the value of $\mathit{extend}(T)([(\beta, j)]) = a$, and the cell at index $[-1, -1]$ is assigned to the value of $\mathit{extend}(T)([]) = a$. This strategy allows us to effectively represent any finite partial map in the target language using a single PyTorch tensor.

Antichains and PyTorch tensor operations. To preserve the correspondence between finite partial maps and PyTorch tensors, our library overrides the standard variable read and write of Python in a way that respects the antichain. An antichain, being a finite set of indices, is itself represented by a PyTorch tensor, where each tensor cell holds a boolean value indicating whether the corresponding index is in the antichain. Based on this representation, our library reads a variable by constructing a PyTorch tensor consisting solely of elements at indices for which the current antichain evaluates to

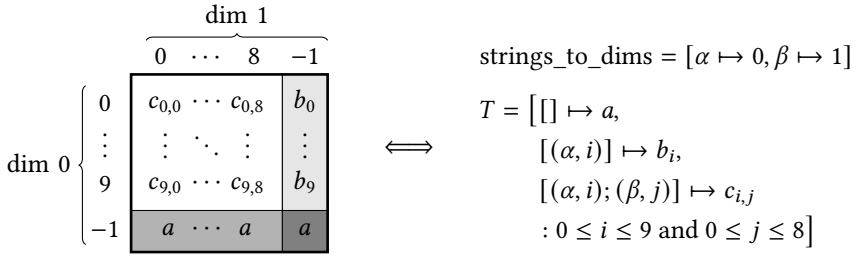

Fig. 10. Correspondence between a PyTorch tensor (left) and a finite partial map T in our target language (right). The strings_to_dims mapping ensures this correspondence by allocating strings to unique dimensions.

true. Subsequent operations (e.g., arithmetic) on PyTorch tensors are then performed element-wise, facilitated by PyTorch's *broadcasting* semantics. The outcomes of these operations are then used to update the PyTorch tensor assigned to the variable, but in a more careful way: it updates not only the element at the current antichain's true indices, but also the elements at the upward closure of those indices, which can be also efficiently handled by broadcasting. These steps are implemented efficiently in PyTorch thanks to its support for broadcasting and to the preservation of the encoding of finite partial maps into PyTorch tensors across all the base operations of the target language.

Variable elimination. Discrete random variables pose difficulties for probabilistic inference algorithms that rely on gradient computation, such as SVI and HMC. A common solution, known as *variable elimination*, enumerates all the possible combinations of the values of these discrete random variables, computes a total score for each combination, and sums them up. While variable elimination is costly in general, Pyro efficiently performs it by allocating a dedicated tensor dimension for the enumeration of each discrete random variable, and leveraging PyTorch's broadcasting semantics and tensor-based parallelism to compute total scores. To support these optimised algorithms, our library internally manages metadata (e.g., dimension-variable mappings) for the correct handling of these additional dimensions for enumeration.

6 Experimental Evaluation

In this section, we report on the evaluation of probabilistic programs translated using our method, focusing on the following three research questions:

(RQ1) Consistency: Does our approach produce scores and their gradients that are consistent with those produced by existing methods (modulo floating-point errors)?
(RQ2) Time and memory: How does our approach compare to existing techniques in terms of execution time and peak memory usage?
(RQ3) Scalability: Does our approach scale with dataset size, assuming a fixed model?

Benchmarks. We used five classes of probabilistic models, with complementary features.

• *Hidden Markov models* (`hmm-{ord1,ord2,neural}`) [10, 47]. These models describe piano scores [12] using variants of a hidden Markov model (HMM). The models `hmm-{ord1,ord2}` are first- and second-order HMMs involving 3 nested loops but no neural networks, whereas `hmm-neural` is a first-order HMM with 2 nested loops and neural networks. In these models, the outermost loop iterates over 229 (`hmm-{ord1,neural}`) or 50 (`hmm-ord2`) sequences of music notes, the first nested loop stands for the HMM iteration over time steps of length 129, and the innermost loop of `hmm-{ord1,ord2}` iterates over piano keys of length 51. The first and third loops are easily parallelisable since each iteration is conditionally independent.

- *Deep Markov model* (`dmm`) [10, 36]. This model is another HMM variant describing the same music dataset. It is a first-order Markov model, consisting of two nested loops over 229 sequences of music notes and 129 time steps, respectively. It differs from `hmm-*` in two ways: first, `dmm` uses continuous hidden states, while `hmm-*` uses discrete states; second, `dmm` uses neural networks for both prior and likelihood, while `hmm-neural` does so only for likelihood.
- *Nested hidden Markov models* (`nhmm-{train,stock}`). These first-order nested HMMs model the number of arrivals at a train station over 44,640 time steps (hours) [64], and the stock price of 10 companies over 2,560 time steps (trading days) [3], respectively. Both models consist of 4 nested loops, either over 5 years, 12 months, 31 days, 24 hours, or over 10 companies, 10 years, 4 quarters, 64 days. Only the outermost loops can be easily parallelised.
- *Autoregressive model* (`arm`). This variant of autoregressive model (AR) [60] describes speech data taken from [49]. While standard AR models have a constant order, the order of this model is a latent random variable. This model contains one loop iterating over 2,000 time steps.
- *Temperature controller model* (`tcm`) [39, 59]. This state-space model describes the evolution of the temperature inside a room, equipped with a stochastic temperature controller. It includes if-then-else statements, required to convey different types of temperature dynamics based on the controller state. It consists of two nested loops over 10 temperature sequences and 200 time steps, respectively, where the first loop can be easily parallelised.

Baseline methods. Pyro features several loop vectorisation methods.

- *Plate constructor [67].* It is a tensor-based loop constructor that vectorises loops without data dependencies. However, it may lead to incorrect inference results if the variables in a loop have data dependencies across iterations. Plates can be nested but cannot contain if-then-else statements.
- *Vectorised Markov constructor [66].* It is a "funsor"-based loop constructor that vectorises certain types of loops with data dependencies. It supports scalar or vector indices determined by the user-defined dependence order, and includes an efficient automatic variable elimination algorithm for enumeration. However, it cannot vectorise if-then-else statements and nested loops.
- *Discrete HMM distribution [65].* It is a distribution class in Pyro that expresses a first-order discrete HMM. Given transition and emission matrices, it efficiently computes log-likelihood and posterior marginals in parallel using algorithms such as the forward-backward algorithm. However, it is limited to first-order discrete HMMs without if-then-else statements or nested loops.
- *Hand-coded vectorisation.* This baseline refers to the manual vectorisation of loops using tensor operations. As a result, applying it requires expertise in both the Pyro PPL and tensor-based programming, and often entails non-standard manipulation of the PPL's internal structures.

We compare our method (`ours`) with five reference methods that combine some of these features:

- `seq`: It is a sequential implementation of the model, where all loops are executed sequentially.
- `plate`: It uses only plate constructors to vectorise loops without data dependencies.
- `vmarkov`: It uses plate constructors and vectorised Markov constructors to vectorise loops.
- `discHMM`: It uses plate constructors and discrete HMM constructors to vectorise loops.
- `hand`: It refers to manual vectorisation of loops using tensor operations.

Note that all methods except `seq` have limitations and cannot be applied to all models. In particular, some benchmark models include if-then-else statements, nested loops, or higher-order dependencies that these methods cannot handle. We consider `plate`, `vmarkov`, or `discHMM` inapplicable to a model if no loop in the model can be vectorised by a plate constructor, a vectorised Markov constructor, or a discrete HMM constructor, respectively. Also, `hand` is considered inapplicable to a model when a loop in the model contains an if-then-else statement. This is because vectorisation using tensor operations in such cases is highly non-trivial. See §B for further details on the applicability. We also note that the use of `plate` or `vmarkov` requires care since they are only sound for loops without

	plate		vmarkov		discHMM		ours	
	obj	grad	obj	grad	obj	grad	obj	grad
hmm-ord1	0	3.5e-7	0	3.5e-7	8.7e-8	2.0e-6	0	3.4e-7
hmm-neural	2.0e-9	4.5e-6	5.0e-9	4.0e-6	4.0e-9	4.1e-6	5.0e-9	4.0e-6
hmm-ord2	0	2.7e-7	0	2.7e-7	-	-	0	2.6e-7
dmm	7.7e-9	3.1e-7	2.5e-8	3.1e-7	-	-	2.4e-8	3.1e-7
nhmm-train	0	8.1e-6	-	-	-	-	0	8.1e-6
nhmm-stock	0	5.4e-6	-	-	-	-	0	5.5e-6
arm	-	-	0	3.0e-9	-	-	0	3.0e-9
tcm	-	-	-	-	-	-	0	0

Table 1. Average relative errors of objective function and gradient values computed by plate, vmarkov, discHMM, and ours with respect to seq in SVI or MAP inference. "-" indicates the method is not applicable. Note that plate is not applicable to arm as it contains only a single loop with data dependencies.

data dependencies or when the dependence order is correctly specified by the user. See §C for the code snippets of the model hmm-neural using our method and the baselines.

Experimental setup. We run three types of inference engines, Maximum a Posteriori (MAP) estimation, Stochastic Variational Inference (SVI), and Markov Chain Monte Carlo (MCMC) sampling. MAP estimation is performed for the models hmm-* and nhmm-*. SVI is run using RNN-based amortised variational distributions for dmm or independent normal variational distributions for arm and tcm. The inference engines for MAP estimation and SVI require solving an optimisation problem, where an objective function is the posterior probability density or the evidence lower bound. We use the Adam optimiser [35] to solve this problem, i.e., to train learnable parameters. MCMC is used for models without neural networks, i.e., hmm-{ord1,ord2}, nhmm-*, arm, and tcm. Specifically, we employ the Hamiltonian Monte Carlo (HMC) sampler [18, 19] with fixed number of leapfrog steps and step size to sample from the posterior distribution of continuous variables, while a discrete variable in arm is handled via Gibbs sampling [26] based on Metropolis-Hastings [28, 41] with a random walk proposal. All timings are measured using an NVIDIA 2080Ti GPU.

[RQ1] Consistency. To validate that our method yields correct outputs, we compared each of the vectorisation methods except hand against seq using SVI or MAP inference, in terms of (i) the objective function value, computed as the sum of score values at each **score** command, and (ii) the gradient of the objective function with respect to learnable parameters computed by automatic differentiation. We then compared these results across the different methods to assess their consistency with seq.

More precisely, we measured the relative errors between these values in L_2-norm. We computed the objective function or gradient values $v \in \mathbb{R}^d$ using seq, and $\hat{v} \in \mathbb{R}^d$ using one of the other vectorisation methods, over the first training step with the same random seed; and then computed the relative error $\|v - \hat{v}\|_2 / \|v\|_2$. To alleviate the high computational cost of the seq method, we used only 10 of the outermost sequences of the datasets used in the hmm-* and dmm benchmarks, while the entire datasets were used for the other benchmarks.

Table 1 presents these relative errors averaged over 100 random seeds for each model and each vectorisation method. It demonstrates that the outputs from ours are the same as, or very close to the outputs from seq for all models considered. We believe that the non-zero errors come from the non-associativity of floating-point addition/multiplication. For objective function values, we manually verified that they are caused by different orderings of floating-point operations: hmm-neural and dmm involve matrix multiplications, and PyTorch computes the addition/multiplication operations there in different orders in ours and seq, due to different tensor dimensions in the computation. For gradient values, we conjecture that a similar reordering of operations is introduced by PyTorch during backpropagation, due to different tensor dimensions or non-contiguous memory layouts. We

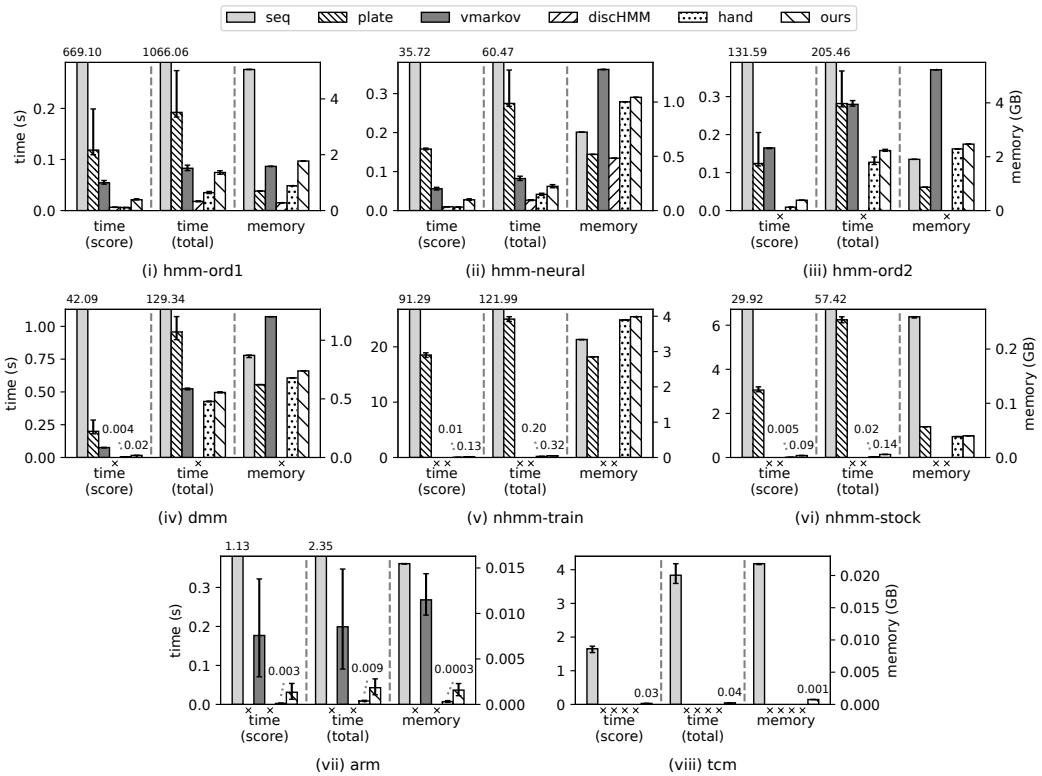

Fig. 11. Score time (s), total time (s), and GPU memory usage (GB) in each training step of SVI/MAP with 95% confidence intervals. "×" below the x-axis indicates inapplicability. Large values are truncated.

also note that similar discrepancies caused by these factors appeared across all other vectorisation methods. Overall, these observations suggest that our method yields consistent outputs, and the remaining errors are well within the bounds of expected floating-point behaviour.

[RQ2] Time and Memory. We compared the wall-clock time and GPU memory usage of our method with other baselines using SVI, MAP inference, and MCMC sampling. For SVI and MAP, we measured, for each training step, the time to compute score (score time) and the time to run the step entirely (total time). The total time includes components other than score computation: e.g., sampling from variational distributions, performing variable elimination, and computing gradients. For MCMC, we measured the time to execute each sampling step. We also measured the maximum GPU memory usage over a training step for SVI and MAP, and over a sampling step for MCMC. For SVI and MAP, every reported number is obtained by averaging over 5 runs, where each run consists of 10 training steps (`seq`) or 1,000 training steps (the others), and the first 1 out of 10 steps (`seq`) or 10 out of 1,000 steps (the others) are excluded from the averages, as considered burn-in steps. Concretely, we averaged over 45 data points for `seq` and 4,950 data points for the other methods. For MCMC, each reported number is the average over 8 runs, each with a different number of sampling steps performed within one hour (i.e., 8 × the average steps per hour in total). In this evaluation, the full datasets are used for all benchmarks. The main results are shown in Fig. 11 and Table 2, the full results are in §D.

Results on SVI/MAP time. Compared to `seq`, `ours` shows significant speedups in both score time (36.2–31399.3×) and total time (54.8–14302.9×). These speedups are mainly enabled by two factors. First, in a score computation, `ours` drastically reduces the iteration counts required for score

computation (2 to 10.1) compared to `seq`'s extensive iterations (2,000 to 1,506,591), achieved by the loop vectorisation with early termination. Second, `ours` consistently employs tensors, while `seq` uses a list of scalars. Many operations in Pyro, including variable elimination and sampling from variational distributions, are more efficient over tensors than separate scalars.

Compared to `plate`, `ours` again shows significant speedups in score time (4.5–145.8×) and total time (1.8–78.4×). These gains are also due to `plate`'s high iteration count (129 to 8,928) and `ours`'s efficient tensor operations. Furthermore, unlike `ours`, `plate` is not applicable to the models `arm` and `tcm`, as all the loops in these models have data dependencies or if-then-else statements.

Compared to `vmarkov`, `ours` is faster in both score time (2.0–6.0×) and total time (1.1–4.6×). These speedups stem from two main sources. First, `ours` exhibits a fewer number of iterations: for a loop with dependence order K, `ours` performs $K + 1$ iterations, while `vmarkov` requires $2K + 1$ iterations. Second, `vmarkov` introduces additional overheads due to the use of "funsors" (functional tensors). A conversion from PyTorch tensors to Pyro funsors, followed by operations over funsors, is much slower than direct operations over tensors. In addition, `vmarkov` is not applicable to the models `nhmm-*` and `tcm`, as they contain nested loops with dependencies or if-then-else statements.

Compared to `discHMM`, `ours` is slower in score time (0.3–0.4×) and total time (0.2–0.4×) for the models `hmm-{ord1,neural}`. However, `discHMM` only supports first-order HMMs without nesting and if-then-else statements, and thus cannot be applied to the other six models.

Compared to `hand`, ours is slower in score time (0.1–0.3×) and total time (0.2–0.9×) for all models except `tcm` where `hand` could not be applied due to the presence of if-then-else statements.

Results on SVI/MAP memory usage. Compared to `seq`, `ours` shows a decrease of memory usage for some models and an increase for others. For models `nhmm-stock`, `arm`, and `tcm`, `ours` significantly reduces peak GPU memory usage, requiring only 3–15% of the memory used by `seq`. For models `hmm-{neural, ord2}` and `nhmm-train`, `ours` requires 119–144% of the memory used by `seq`. Two effects play against each other here. Our approach uses antichains representing indices considered in a given state. In our current implementation, these antichains are represented as boolean tensors, which may be sparse, hence causing memory inefficiencies. However, even then, `ours` may benefit from more efficient memory allocation schemes implemented by GPUs, e.g., with allocation in large fixed-sized chunks. In contrast, `seq` allocates larger numbers of smaller chunks, which may increase memory overhead, especially when the maximum vector dimension used in the innermost loop is small. These factors explain that `ours` can sometimes achieve lower memory usage under specific conditions, although it seems to have a higher memory requirement in theory.

Compared to `plate`, `ours` reduces memory usage to 70% on `nhmm-stock` benchmark, due to the trade-off between usage of antichains and memory allocation schemes discussed above. On the other models, `ours` uses more memory, ranging from 119% to 284% of the memory used by `plate`.

Compared to `vmarkov`, `ours` uses less memory in all models except `hmm-ord1`. As we remarked above, `ours` stores antichains which require additional memory. However, `vmarkov` also induces an important memory overhead, due to additional execution of iterations with scalar indices and the implementation of funsors. The results demonstrate that the overall overhead due to funsor in `vmarkov` is higher than that generated by antichains in `ours`.

Compared to `discHMM`, `ours` uses more memory for the two models that the former can handle. However, as mentioned earlier, `discHMM` is specialised to a narrow class of models, so it could not be applied to the other six models.

Compared to `hand`, `ours` uses more memory for all models except `tcm` where `hand` could not be applied. Specifically, `ours` uses 518% and 200% of the memory used by `hand` on `arm` and `hmm-ord1`. For the others, `ours` uses 102–109% of the usage of `hand`, showing comparable memory usage.

	seq	plate	vmarkov	discHMM	hand
MCMC speedup (other → ours)	67–1210×	1.83–65.63×	1.07–2.56×	0.28×	0.26–0.80×
MCMC memory usage (ours/other)	3–120%	69–253%	24–109%	562%	102–1290%

Table 2. Speedup and and relative GPU memory usage per MCMC sampling step, compared between ours and the others. Each entry shows the range over applicable benchmarks. See §D for full results.

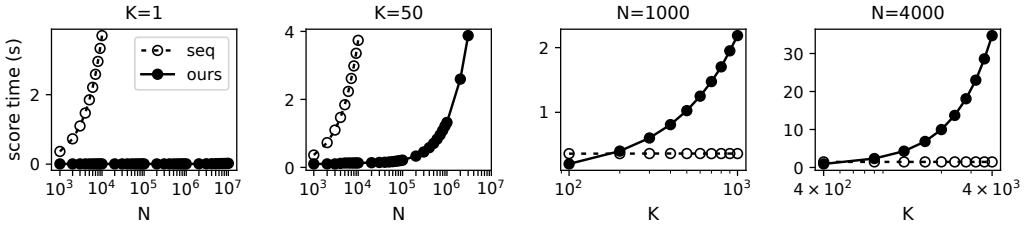

Fig. 12. Score time (s) of ours and seq for the model arm' with different N and K. Each point denotes the mean over 99 runs. 95% confidence intervals are omitted as their relative width to the mean is below 2%.

Results on MCMC. As shown in Table 2, the results are consistent with the observations from the SVI and MAP inference experiments across all benchmarks, except for hmm-neural and dmm benchmarks that we do not consider in MCMC experiments. Overall, ours exhibits superior sampling speed (i.e., lower time per sampling step) and broader applicability compared to the vectorisation baselines, often while maintaining comparable or lower memory consumption. For instance, compared to vmarkov, ours performs 1.07–2.56× more sampling steps, while using 24–109% of the GPU memory consumed by vmarkov, across all benchmarks except nhmm-* and tcm, for which vmarkov is inapplicable. The applicability of each method to the benchmarks is consistent with that observed in the SVI and MAP inference experiments, except for the case of seq on hmm-ord1, where it fails to complete a single step within one hour.

[RQ3] Scalability. To assess the scalability of our method, we considered the autoregressive model arm', where N denotes the number of samples and K the order of the model: $y_i \sim \mathcal{N}(y_{i-1} + \cdots + y_{i-K}, 1)$ for $i = 0, \ldots, N-1$. Here y_i is the i-th sample for $i = 0, \ldots, N-1$, and the initial values are $y_{-K} = \cdots = y_{-1} = 0$. We express the model in Pyro using a loop of length N to generate the y_i's, which takes $K+1$ iterations to reach a fixed point when using our method.

We measured the time to compute score (score time) under two setups: use a fixed $K \in \{1, 50\}$ and vary N; and use a fixed $N \in \{1000, 4000\}$ and vary K. All samples were generated from the standard normal distribution, and the time for generating the samples was excluded from the measurement. For each (N, K) pair, we averaged the time over 99 runs out of 100, excluding the first burn-in step. We compared the time taken by the vectorised loop produced by our method (ours) against the sequential loop (seq). The results are shown in Fig. 12.

When K is fixed to a value significantly smaller than N (i.e., 1 and 50), ours exhibits a notably slower increase rate with respect to N compared to seq. The improvement is more significant when K is small. This demonstrates that, when the dependence order of the loop is low, our vectorisation method can improve performance more substantially by leveraging the GPU parallelism.

In contrast, when N is fixed (i.e., 1000 and 4000) and K is close to N, the runtime of seq remains almost constant, while that of ours is quadratic in K. Indeed, when K and N are close to each other, both the number of iterations and the size of the vectors processed within each iteration grow proportionally to K. These findings suggest that our method may not be optimal for higher-order loops. Nevertheless, in many practical applications of autoregressive models, such as speech processing [8, 60], heart rate variability analysis [11], and image dynamic texture modelling [31],

the value of K is typically much smaller than N. To mitigate the performance degradation when K approaches N, an adaptive strategy could be employed to switch between sequential and vectorised execution based on the loop dependence order (e.g., K), which our method can automatically detect.

7 Related Work

Model translations in PPLs. Several prior approaches have translated models from non-tensor-based PPLs into tensor-based languages, such as compiling Stan [17] models to Pyro or NumPyro models for improved inference performance with provable correctness [6], or converting high-level discrete probabilistic programs into NumPy [69] programs with optimised tensor operations (e.g., `opt_einsum` [58]) for exact inference over discrete random variables [48]. These approaches do not alter the structures of the original models as aggressively as our method does for loops; instead, they mostly rely on generic optimisation of the target language for performance improvements.

Vectorisation primitives in PPLs. Most tensor-based PPLs have introduced convenient features to vectorise independent operations in probabilistic models. A key example is the `plate` primitive in Pyro and NumPyro, which originates from graphical models [15] and is used in PPLs to vectorise loops, particularly when the random variables inside the loop are conditionally independent or when the loop has no data dependencies. Similarly, JAX provides `vmap` [13] and TensorFlow offers `vectorized_map` [2], both of which aid in vectorising function evaluation over independent tensor dimensions. These primitives also help vectorise MCMC sampling, improving inference efficiency across multiple chains [37]. While these primitives are effective for simple loops or sequences of independent operations, our work extends this capability to loops with more complex dependencies.

Other works in PPL aim at extending vectorisation primitives to handle more complex dependencies, similarly to our approach. For instance, TensorFlow Probability (TFP) [23] provides `MarkovChain` distributions in order to vectorise loops with short-range dependencies in models such as random walks or autoregressive models. Similarly, Pyro provides a family of hidden Markov model (HMM) distributions (e.g., `DiscreteHMM`, `GaussianHMM`) to vectorise key computations in HMMs, such as smoothing, filtering algorithms, and score computation [56]. However, these approaches usually require the restructuring of a model so as to make explicit use of specific vectorisation primitives, or have limitations related to the kind of dependencies they support (e.g., supporting only predefined transition distributions). More closely related to our work, Funsor [46] introduces `vectorized_markov` to vectorise loops with dependencies, but requires users to manually specify dependency lengths, and is not generally applicable to all dependency structures (e.g., nested structures or conditional branches). Our work extends these ideas by introducing a more general, automatic vectorisation method that ensures correctness through fixed-point check and eliminates the need for any additional user input or extensive model change. Moreover, it can handle complex dependencies, which may be nested or have dependencies determined at runtime.

Optimisation of inference algorithms. Our work is related to approaches that optimise parameter learning or posterior inference algorithms by exploiting properties of PPL models. A static analysis for the smoothness properties of PPL models has been developed to propose an SVI algorithm that exploits these properties to reduce variance in gradient estimation [38]. Other work focuses on conditional independence properties in PPL models and proposes a type system for capturing these properties, along with an optimised variable elimination algorithm that leverages them [27]. Meanwhile, methods have been introduced to automatically derive correct and efficient gradient estimators for a given SVI objective when it is expressed as a program, by transforming it into gradient estimation code [7]. Our method does not directly optimise learning or inference algorithms; instead, it optimises the execution of probabilistic programs by vectorising loops with dependencies, which in turn can accelerate various inference algorithms relying on efficient program execution.

Automatic vectorisation in general-purpose languages. Our work is also related to studies in general-purpose programming languages that aim at automating vectorisation. For example, dynamic data dependency analysis techniques have been proposed to infer SIMD parallelisation potential by identifying all possible dependency-preserving reorderings, together with further analysis of memory access patterns [30]. Additionally, methods have been developed to automatically batch operations in dynamic neural network libraries [44], such as DyNet [43], and to handle data-dependent control flow and recursion in programs using a pointer per batch element to manage variables and program counters [54]. PyTorch's framework for automatic batching [14] leverages mask propagation and control flow rewriting to simplify minibatching in dynamic models. Recently, Autovesk [61] automates vectorisation by transforming scalar instructions into vectorised instructions over a graph of operations, using heuristics to minimise the overall instruction numbers. Although these methods rely on static or dynamic analysis of data dependencies to exploit vectorisation opportunities, our work avoids the need for complex analysis by an automated fixed-point check that ensures correctness of vectorisation.

There are also approaches that use thread-level speculation (TLS) to parallelise general sequential programs. R-LRPD [21], for instance, speculatively executes loop iterations in parallel and records read/write bit arrays to re-execute dependent iterations. GPU-TLS [71] and STMlite [40] both focus on re-execution of mis-speculated iterations, and dynamic analysis to ensure correctness. Other methods improve performance by integrating TLS with transactional memory (TM) [5] or adopting in-place speculation [45]. However, these methods typically discard partial results on mis-speculation during re-execution, which can cause a large number of re-executions and slow convergence, particularly in loops with many inter-iteration dependencies, as observed in most of the models in our experiments. In contrast, our method can extract useful partial information even from such mis-speculated iterations through the **shift** construct, achieving the faster convergence in those cases, as demonstrated by our experimental results.

8 Conclusion and Limitations

We have presented an automatic sound method for the vectorisation of loops over a fixed range, as often found in PPLs. It is based on a speculative execution of the body of for-loops, using a non-standard semantics, and on a fixed-point check to ensure correctness of the final result. Its implementation upon Pyro proceeds by translation into a lower level PPL with primitives for vectorisation. We proved this correctness of the method and demonstrated its effectiveness in accelerating inference for several families of complex probabilistic models including sequence models and hierarchical models, and for various probabilistic inference engines.

While our work leads to performance improvements in our experiments, it also has limitations. First, its fixed-point check introduces a performance overhead that might not always be compensated for when the iteration count is not substantially reduced by our method—for instance, due to long data dependencies, as discussed in §6. Although such cases are not expected to be dominant in practice, a potential avenue for improvement lies in exploring a hybrid approach that dynamically decides whether to apply vectorising optimisation or fall back to a standard runtime based on program characteristics. Second, our work does not consider the vectorisation of sample generation, which is central to many inference algorithms (e.g., importance sampling, Metropolis-Hastings, and Sequential Monte Carlo [22]). Our method instead focuses on the vectorisation of density computations, which is orthogonal to sample generation and is central to other inference algorithms (e.g., SVI and HMC). Addressing this limitation would yield a more comprehensive vectorisation framework and further accelerate a broader class of inference algorithms.

References

[1] Oriol Abril-Pla, Virgile Andreani, Colin Carroll, Larry Dong, Christopher J Fonnesbeck, Maxim Kochurov, Ravin Kumar, Junpeng Lao, Christian C Luhmann, Osvaldo A Martin, et al. 2023. PyMC: a modern, and comprehensive probabilistic programming framework in Python. *PeerJ Computer Science* 9 (2023), e1516. doi:10.7717/peerj-cs.1516

[2] Ashish Agarwal. 2019. Static automatic batching in TensorFlow. In *International Conference on Machine Learning (ICML)*. 92–101. https://proceedings.mlr.press/v97/agarwal19a.html

[3] Ran Aroussi. 2020. yfinance: Download market data from Yahoo! Finance's API. https://github.com/ranaroussi/yfinance Accessed: 2025-07-10.

[4] Sudipto Banerjee, Bradley P Carlin, and Alan E Gelfand. 2003. *Hierarchical Modeling and Analysis for Spatial Data*. Chapman and Hall/CRC. doi:10.1201/9780203487808

[5] Joao Barreto, Aleksandar Dragojevic, Paulo Ferreira, Ricardo Filipe, and Rachid Guerraoui. 2012. Unifying thread-level speculation and transactional memory. In *International Middleware Conference (Middleware)*. 187–207. doi:10.1007/978-3-642-35170-9_10

[6] Guillaume Baudart, Javier Burroni, Martin Hirzel, Louis Mandel, and Avraham Shinnar. 2021. Compiling Stan to generative probabilistic languages and extension to deep probabilistic programming. In *ACM Conference on Programming Language Design and Implementation (PLDI)*. 497–510. doi:10.1145/3453483.3454058

[7] McCoy R. Becker, Alexander K. Lew, Xiaoyan Wang, Matin Ghavami, Mathieu Huot, Martin C. Rinard, and Vikash K. Mansinghka. 2024. Probabilistic Programming with Programmable Variational Inference. *Proc. ACM Program. Lang.* 8, PLDI, Article 233 (June 2024), 25 pages. doi:10.1145/3656463

[8] Maria A Berezina, Daniel Rudoy, and Patrick J Wolfe. 2010. Autoregressive modeling of voiced speech. In *International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. 5042–5045. doi:10.1109/ICASSP.2010.5495058

[9] Eli Bingham, Jonathan P. Chen, Martin Jankowiak, Fritz Obermeyer, Neeraj Pradhan, Theofanis Karaletsos, Rohit Singh, Paul A. Szerlip, Paul Horsfall, and Noah D. Goodman. 2019. Pyro: Deep Universal Probabilistic Programming. *Journal of Machine Learning Research* 20, 28 (2019), 1–6. http://jmlr.org/papers/v20/18-403.html

[10] Eli Bingham, Jonathan P. Chen, Martin Jankowiak, Fritz Obermeyer, Neeraj Pradhan, Theofanis Karaletsos, Rohit Singh, Paul A. Szerlip, Paul Horsfall, and Noah D. Goodman. 2019. Pyro: Deep Universal Probabilistic Programming with Python and PyTorch. https://github.com/pyro-ppl/pyro Accessed: 2025-07-10.

[11] Anita Boardman, Fernando Soares Schlindwein, and Ana Paula Rocha. 2002. A Study on The Optimum Order of Autoregressive Models for Heart Rate Variability. *Physiological Measurement* 23, 2 (2002), 325. doi:10.1088/0967-3334/23/2/308

[12] Nicolas Boulanger-Lewandowski, Yoshua Bengio, and Pascal Vincent. 2012. Modeling temporal dependencies in high-dimensional sequences: application to polyphonic music generation and transcription. In *International Conference on Machine Learning (ICML)*. 1881–1888. https://icml.cc/2012/papers/590.pdf

[13] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. 2018. *JAX: composable transformations of Python+NumPy programs*. http://github.com/jax-ml/jax Accessed: 2025-07-10.

[14] James Bradbury and Chunli Fu. 2018. Automatic batching as a compiler pass in PyTorch. In *Workshop on Systems for ML at NeurIPS 2018*. http://learningsys.org/nips18/assets/papers/107CameraReadySubmissionMatchbox__LearningSys_Abstract_%20(2).pdf

[15] Wray L Buntine. 1994. Operations for learning with graphical models. *Journal of Artificial Intelligence Research* 2 (1994), 159–225. doi:10.1613/jair.62

[16] Ricardo Cannizzaro, Michael Groom, Jonathan Routley, Robert Osazuwa Ness, and Lars Kunze. 2025. COBRA-PPM: A Causal Bayesian Reasoning Architecture Using Probabilistic Programming for Robot Manipulation Under Uncertainty. *arXiv:2403.14488* (2025). doi:10.48550/arXiv.2403.14488

[17] Bob Carpenter, Andrew Gelman, Matthew Hoffman, Daniel Lee, Ben Goodrich, Michael Betancourt, Marcus Brubaker, Jiqiang Guo, Peter Li, and Allen Riddell. 2017. Stan: A Probabilistic Programming Language. *Journal of Statistical Software* 76, 1 (2017), 1–32. doi:10.18637/jss.v076.i01

[18] Wenlin Chen, Mingtian Zhang, Brooks Paige, José Miguel Hernández-Lobato, and David Barber. 2024. Diffusive Gibbs Sampling. In *International Conference on Machine Learning*. PMLR.

[19] Wenlin Chen, Mingtian Zhang, Brooks Paige, José Miguel Hernández-Lobato, and David Barber. 2024. *Diffusive Gibbs Sampling*. https://github.com/Wenlin-Chen/DiGS Accessed: 2025-10-20.

[20] Peter D Congdon. 2019. *Bayesian Hierarchical Models: With Applications Using R*. Chapman and Hall/CRC. doi:10.1201/9780429113352

[21] F. Dang, Hao Yu, and L. Rauchwerger. 2002. The R-LRPD test: speculative parallelization of partially parallel loops. In *International Parallel and Distributed Processing Symposium (IPDPS)*. doi:10.1109/IPDPS.2002.1015493

[22] Pierre Del Moral, Arnaud Doucet, and Ajay Jasra. 2006. Sequential Monte Carlo Samplers. *Journal of the Royal Statistical Society Series B: Statistical Methodology* 68, 3 (05 2006), 411–436. doi:10.1111/j.1467-9868.2006.00553.x

[23] Joshua V Dillon, Ian Langmore, Dustin Tran, Eugene Brevdo, Srinivas Vasudevan, Dave Moore, Brian Patton, Alex Alemi, Matt Hoffman, and Rif A Saurous. 2017. Tensorflow distributions. *arXiv:1711.10604* (2017). doi:10.48550/arXiv.1711.10604

[24] Simon Duane, A.D. Kennedy, Brian J. Pendleton, and Duncan Roweth. 1987. Hybrid Monte Carlo. *Physics Letters B* 195, 2 (1987), 216–222. doi:10.1016/0370-2693(87)91197-X

[25] Hong Ge, Kai Xu, and Zoubin Ghahramani. 2018. Turing: a language for flexible probabilistic inference. In *International Conference on Artificial Intelligence and Statistics (AISTATS)*. 1682–1690. http://proceedings.mlr.press/v84/ge18b.html

[26] Stuart Geman and Donald Geman. 1984. Stochastic relaxation, Gibbs distributions, and the Bayesian restoration of images. *IEEE Transactions on pattern analysis and machine intelligence* 6 (1984), 721–741.

[27] Maria I. Gorinova, Andrew D. Gordon, Charles Sutton, and Matthijs Vákár. 2021. Conditional Independence by Typing. *ACM Trans. Program. Lang. Syst.* 44, 1, Article 4 (Dec. 2021), 54 pages. doi:10.1145/3490421

[28] W Keith Hastings. 1970. Monte Carlo sampling methods using Markov chains and their applications. (1970).

[29] Matthew D. Hoffman, David M. Blei, Chong Wang, and John Paisley. 2013. Stochastic variational inference. *J. Mach. Learn. Res.* 14, 1 (2013), 1303–1347. https://jmlr.org/papers/v14/hoffman13a.html

[30] Justin Holewinski, Ragavendar Ramamurthi, Mahesh Ravishankar, Naznin Fauzia, Louis-Noël Pouchet, Atanas Rountev, and P Sadayappan. 2012. Dynamic trace-based analysis of vectorization potential of applications. In *ACM Conference on Programming Language Design and Implementation (PLDI)*. 371–382. doi:10.1145/2254064.2254108

[31] Midori Hyndman, Allan D Jepson, and David J Fleet. 2007. Higher-order Autoregressive Models for Dynamic Textures. In *British Machine Vision Conference (BMVC)*. doi:10.5244/C.21.76

[32] Krishna Murthy Jatavallabhula, Miles Macklin, Dieter Fox, Animesh Garg, and Fabio Ramos. 2023. Bayesian Object Models for Robotic Interaction with Differentiable Probabilistic Programming. In *Conference on Robot Learning (CoRL)*. 1563–1574. https://proceedings.mlr.press/v205/jatavallabhula23a.html

[33] Marc Kéry and Kenneth F Kellner. 2024. *Applied Statistical Modelling for Ecologists: A Practical Guide to Bayesian and Likelihood Inference Using R, JAGS, NIMBLE, Stan and TMB*. Elsevier. doi:10.1016/C2022-0-02766-5

[34] Jack Kiefer and Jacob Wolfowitz. 1952. Stochastic estimation of the maximum of a regression function. *The Annals of Mathematical Statistics* (1952), 462–466. doi:10.1214/aoms/1177729392

[35] Diederik P. Kingma and Jimmy Ba. 2015. Adam: A Method for Stochastic Optimization. In *International Conference on Learning Representations (ICLR)*. doi:10.48550/arXiv.1412.6980

[36] Rahul Krishnan, Uri Shalit, and David Sontag. 2017. Structured Inference Networks for Nonlinear State Space Models. In *AAAI Conference on Artificial Intelligence (AAAI)*. doi:10.1609/aaai.v31i1.10779

[37] Junpeng Lao, Christopher Suter, Ian Langmore, Cyril Chimisov, Ashish Saxena, Pavel Sountsov, Dave Moore, Rif A Saurous, Matthew D Hoffman, and Joshua V Dillon. 2020. tfp.mcmc: Modern Markov chain Monte Carlo tools built for modern hardware. *arXiv:2002.01184* (2020). doi:10.48550/arXiv.2002.01184

[38] Wonyeol Lee, Xavier Rival, and Hongseok Yang. 2023. Smoothness Analysis for Probabilistic Programs with Application to Optimised Variational Inference. *Proc. ACM Program. Lang.* 7, POPL, Article 12 (Jan. 2023), 32 pages. doi:10.1145/3571205

[39] Wonyeol Lee, Hangyeol Yu, and Hongseok Yang. 2018. Reparameterization Gradient for Non-differentiable Models. In *Annual Conference on Neural Information Processing Systems (NeurIPS)*. https://dl.acm.org/doi/10.5555/3327345.3327459

[40] Mojtaba Mehrara, Jeff Hao, Po-Chun Hsu, and Scott Mahlke. 2009. Parallelizing sequential applications on commodity hardware using a low-cost software transactional memory. In *ACM Conference on Programming Language Design and Implementation (PLDI)*. 166–176. doi:10.1145/1542476.1542495

[41] Nicholas Metropolis, Arianna W Rosenbluth, Marshall N Rosenbluth, Augusta H Teller, and Edward Teller. 1953. Equation of state calculations by fast computing machines. *The journal of chemical physics* 21, 6 (1953), 1087–1092.

[42] Radford M Neal et al. 2011. MCMC using Hamiltonian dynamics. *Handbook of Markov Chain Monte Carlo* 2, 11 (2011), 2. doi:10.1201/b10905

[43] Graham Neubig, Chris Dyer, Yoav Goldberg, Austin Matthews, Waleed Ammar, Antonios Anastasopoulos, Miguel Ballesteros, David Chiang, Daniel Clothiaux, Trevor Cohn, Kevin Duh, Manaal Faruqui, Cynthia Gan, Dan Garrette, Yangfeng Ji, Lingpeng Kong, Adhiguna Kuncoro, Gaurav Kumar, Chaitanya Malaviya, Paul Michel, Yusuke Oda, Matthew Richardson, Naomi Saphra, Swabha Swayamdipta, and Pengcheng Yin. 2017. DyNet: The Dynamic Neural Network Toolkit. *arXiv:1701.03980* (2017). doi:10.48550/arXiv.1701.03980

[44] Graham Neubig, Yoav Goldberg, and Chris Dyer. 2017. On-the-fly operation batching in dynamic computation graphs. In *Annual Conference on Neural Information Processing Systems (NeurIPS)*. https://dl.acm.org/doi/10.5555/3294996.3295153

[45] Cosmin E Oancea, Alan Mycroft, and Tim Harris. 2009. A lightweight in-place implementation for software thread-level speculation. In *Annual Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 223–232. doi:10.1145/1583991.1584050

[46] Fritz Obermeyer, Eli Bingham, Martin Jankowiak, Du Phan, and Jonathan Chen. 2020. Functional Tensors for Probabilistic Programming. *arXiv:1910.10775* (2020). doi:10.48550/arXiv.1910.10775

[47] Fritz Obermeyer, Eli Bingham, Martin Jankowiak, Neeraj Pradhan, Justin Chiu, Alexander Rush, and Noah Goodman. 2019. Tensor Variable Elimination for Plated Factor Graphs. In *International Conference on Machine Learning (ICML)*, Vol. 97. 4871–4880. https://proceedings.mlr.press/v97/obermeyer19a.html

[48] Jingwen Pan and Amir Shaikhha. 2023. Compiling discrete probabilistic programs for vectorized exact inference. In *International Conference on Compiler Construction (CC)*. 13–24. doi:10.1145/3578360.3580258

[49] Vassil Panayotov, Guoguo Chen, Daniel Povey, and Sanjeev Khudanpur. 2015. Librispeech: an Asr Corpus based on Public Domain Audio Books. In *International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. 5206–5210. doi:10.1109/ICASSP.2015.7178964

[50] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Z. Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Annual Conference on Neural Information Processing Systems (NeurIPS)*. 8024–8035. https://dl.acm.org/doi/10.5555/3454287.3455008

[51] Du Phan, Neeraj Pradhan, and Martin Jankowiak. 2019. Composable Effects for Flexible and Accelerated Probabilistic Programming in NumPyro. *arXiv:1912.11554* (2019). doi:10.48550/arXiv.1912.11554

[52] Matti Pärssinen, Mikko Kotila, and Ilkka Sillanpää. 2021. Probabilistic Programming Method for Time-Series Forecasting of COVID-19 Cases Based on Empirical Data. *American Journal of Epidemiology and Infectious Disease* 9 (2021), 18–23. doi:10.12691/ajeid-9-1-4

[53] Song S Qian, Mark R DuFour, and Ibrahim Alameddine. 2022. *Bayesian Applications in Environmental and Ecological Studies with R and Stan*. Chapman and Hall/CRC. doi:10.1201/9781351018784

[54] Alexey Radul, Brian Patton, Dougal Maclaurin, Matthew Hoffman, and Rif A. Saurous. 2020. Automatically batching control-intensive programs for modern accelerators. In *Machine Learning and Systems (MLSys)*, Vol. 2. 390–399. https://proceedings.mlsys.org/paper_files/paper/2020/file/39945d578f616735572174bf5e8f155d-Paper.pdf

[55] Herbert Robbins and Sutton Monro. 1951. A stochastic approximation method. *The Annals of Mathematical Statistics* (1951), 400–407. doi:10.1214/aoms/1177729586

[56] Simo Särkkä and Ángel F García-Fernández. 2020. Temporal parallelization of Bayesian smoothers. *IEEE Trans. Automat. Control* 66, 1 (2020), 299–306. doi:10.1109/TAC.2020.2976316

[57] Viktor Senderov, Jan Kudlicka, Daniel Lundén, Viktor Palmkvist, Mariana P Braga, Emma Granqvist, David Broman, and Fredrik Ronquist. 2023. TreePPL: a universal probabilistic programming language for phylogenetics. *bioRxiv* (2023), 2023–10. doi:10.1101/2023.10.10.561673

[58] Daniel G. A. Smith and Johnnie Gray. 2018. opt_einsum - A Python package for optimizing contraction order for einsum-like expressions. *Journal of Open Source Software* 3, 26 (2018), 753. doi:10.21105/joss.00753

[59] Sadegh Esmaeil Zadeh Soudjani, Rupak Majumdar, and Tigran Nagapetyan. 2017. Multilevel Monte Carlo Method for Statistical Model Checking of Hybrid Systems. In *International Conference on Quantitative Evaluation of Systems (QEST)*. 351–367. doi:10.1007/978-3-319-66335-7_24

[60] Gintautas Tamulevičius and Jonas Kaukenas. 2017. High-order Autoregressive Modeling of Individual Speaker's Qualities. In *IEEE Workshop on Advances in Information, Electronic and Electrical Engineering (AIEEE)*. doi:10.1109/AIEEE.2017.8270551

[61] Hayfa Tayeb, Ludovic Paillat, and Bérenger Bramas. 2023. Autovesk: Automatic Vectorized Code Generation from Unstructured Static Kernels Using Graph Transformations. *ACM Transactions on Architecture and Code Optimization* 21 (2023), 1–25. doi:10.1145/3631709

[62] David Tolpin, Jan-Willem van de Meent, Hongseok Yang, and Frank Wood. 2016. Design and Implementation of Probabilistic Programming Language Anglican. In *Symposium on the Implementation and Application of Functional Programming Languages (IFL)*. Article 6. doi:10.1145/3064899.3064910

[63] Dustin Tran, Matthew D. Hoffman, Dave Moore, Christopher Suter, Srinivas Vasudevan, Alexey Radul, Matthew Johnson, and Rif A. Saurous. 2018. Simple, Distributed, and Accelerated Probabilistic Programming. In *Annual Conference on Neural Information Processing Systems (NeurIPS)*. https://dl.acm.org/doi/10.5555/3327757.3327859

[64] Bay Area Rapid Transit. 2024. Ridership Reports. https://www.bart.gov/about/reports/ridership Accessed: 2025-07-10.

[65] Uber Technologies, Inc. 2025. Pyro Documentation: Distributions (DiscreteHMM). https://docs.pyro.ai/en/dev/distributions.html#pyro.distributions.DiscreteHMM Accessed: 2025-07-10.

[66] Uber Technologies, Inc. 2025. Pyro Documentation: Funsor-Based Pyro (vectorized_markov). https://docs.pyro.ai/en/dev/contrib.funsor.html#pyro.contrib.funsor.vectorized_markov Accessed: 2025-07-10.

[67] Uber Technologies, Inc. 2025. Pyro Documentation: Primitives (plate). https://docs.pyro.ai/en/dev/primitives.html#pyro.primitives.plate Accessed: 2025-07-10.

[68] Stéfan van der Walt, S. Chris Colbert, and Gaël Varoquaux. 2011. The NumPy Array: A Structure for Efficient Numerical Computation. *Computing in Science & Engineering* 13, 2 (2011), 22–30. doi:10.1109/MCSE.2011.37

[69] Stefan van der Walt, S. Chris Colbert, and Gael Varoquaux. 2011. The NumPy Array: A Structure for Efficient Numerical Computation. *Computing in Science & Engineering* 13, 2 (2011), 22–30. doi:10.1109/MCSE.2011.37

[70] Frank Wood, Andrew Warrington, Saeid Naderiparizi, Christian Weilbach, Vaden Masrani, William Harvey, Adam Ścibior, Boyan Beronov, John Grefenstette, Duncan Campbell, and S. Ali Nasseri. 2022. Planning as Inference in Epidemiological Dynamics Models. *Frontiers in Artificial Intelligence* 4 (2022). doi:10.3389/frai.2021.550603

[71] Chenggang Zhang, Guodong Han, and Cho-Li Wang. 2013. GPU-TLS: An Efficient Runtime for Speculative Loop Parallelization on GPUs. In *International Symposium on Cluster, Cloud, and Grid Computing (CCGrid)*. 120–127. doi:10.1109/CCGrid.2013.34

[72] Walter Zucchini and Iain L MacDonald. 2009. *Hidden Markov Models for Time Series: An Introduction Using R.* Chapman and Hall/CRC. doi:10.1201/9781420010893

Received 10 July 2025; revised 23 October 2025

A Proof of Soundness of Parallelising Translation

In this section, we prove Theorem 1, the soundness theorem for our parallelising translation in §3.3. The re-stated corollary and its complete proof can be found in Corollary 5. The proof of the corollary proceeds in the following steps:

(1) We define a variant of the evaluation rules in the target language, called the *intermediate evaluation relation*, denoted $\Downarrow_{int}$, and prove that it preserves the meaning of the original evaluation relation $\Downarrow_{tgt}$ in the target language (§A.1, intermediate-to-target).
(2) Changing only the types of variables of commands in the source language so that the commands belong to the target language preserves the semantics under evaluation using $\Downarrow$ and $\Downarrow_{int}$ (§A.3, source-to-intermediate).
(3) An optimisation of target language commands from the second step, which replaces for-loops with indexless loops under the parallelising mechanism, preserves the semantics under evaluation using $\Downarrow_{int}$ (§A.4, intermediate-to-intermediate).
(4) From the observations in preceding steps, our parallelising translation from the source language to the target language preserves the semantics under evaluation using $\Downarrow$ and $\Downarrow_{tgt}$ (§A.5, source-to-target).

The last three steps rely on a class of binary relations called *couplings*, whose definition and properties are described in §A.2.

A.1 New Evaluation Rules for Proof of Soundness

In this section, we introduce a variant of the evaluation rules in the target language defined in §3. This variant will play a crucial role for formally proving the soundness of our parallelising translation in the following sections. First, we define an *intermediate evaluation relation*, denoted as $\Downarrow_{int}$ and defined on the target language commands, with which the evaluation of **loop_fixpt_noacc** command does not terminate by fixed-point check, but repeats loops for a fixed number of iterations. Then, we prove that the new evaluation rules defined by $\Downarrow_{int}$ do not change the meaning of commands specified by the original evaluation relation $\Downarrow_{tgt}$.

The semantics of **loop_fixpt_noacc**($\mathbf{n}_+$) **C** in the target language enables the optimisation of the execution of the loop using a fixed point check. The result score of (**loop_fixpt_noacc**(n_+) C) entirely depends on the last execution of the body C. Once the loop body C no longer changes the state, repeating its execution n_+ times has no further effect: the state remains unchanged until the loop terminates, and the result score remains the same as in the last execution. We formalise this by defining an intermediate evaluation relation $\Downarrow_{int}$, a variant of $\Downarrow_{tgt}$. It is defined on the same semantic domains and structures as $\Downarrow_{tgt}$:

$$\Downarrow_{int} \subseteq (Com_{tgt} \times RDB \times State_{tgt} \times FAChain) \times (State_{tgt} \times Tensor_{\mathbb{R}}),$$

but only replaces the original evaluation rule for the **loop_fixpt_noacc** command with the following:

$$\frac{\sigma_0 = \sigma \qquad C, D, \sigma_k, A \Downarrow_{int} \sigma_{k+1}, T_{k+1} \text{ for all } k \in \{0, \ldots, n_+ - 1\}}{(\textbf{loop\_fixpt\_noacc}(n_+)\ C), D, \sigma, A \Downarrow_{int} \sigma_{n_+}, T_{n_+}},$$

while preserving the other rules of $\Downarrow_{tgt}$. The next theorem shows $\Downarrow_{int}$ does not change the meaning of commands specified by the original $\Downarrow_{tgt}$ relation.

Theorem 2. *For all commands C in the target language, RDBs D, states $\sigma, \sigma' \in State_{tgt}$, antichains A, and real tensors $T \in Tensor_{\mathbb{R}}$, we have*

$$C, D, \sigma, A \Downarrow_{tgt} \sigma', T \quad \textit{if and only if} \quad C, D, \sigma, A \Downarrow_{int} \sigma', T.$$

Proof. We denote $n = n_+$ for simplicity. We will prove the only-if direction first. The proof proceeds by induction on the depth of the derivation of $C, D, \sigma, A \Downarrow_{tgt} \sigma', T$. Since the rules of $\Downarrow_{tgt}$ and $\Downarrow_{int}$ only differ in **loop_fixpt_noacc**, it is enough to prove the case where the last derivation step applies the rule for **loop_fixpt_noacc** (the proofs for the other cases are trivial).

When the last derivation is the second rule for **loop_fixpt_noacc**, the premises for the last derivation already has all premises for the rule of **loop_fixpt_noacc** in $\Downarrow_{int}$. Therefore, we can trivially derive the same relation through the original rule. When the last derivation is the first rule, the premises for the last derivation are as follow:

$$0 \le m < n - 1 \qquad \sigma_0 = \sigma$$
$$C, D, \sigma_k, A \Downarrow_{tgt} \sigma_{k+1}, T_{k+1} \text{ and } \sigma_k \neq \sigma_{k+1} \text{ for all } 0 \le k \le m-1 \qquad C, D, \sigma_m, A \Downarrow_{tgt} \sigma_m, T_{m+1},$$

where the inferred relation is as follows:

$$(\textbf{loop\_fixpt\_noacc}(n)\ C), D, \sigma, A \Downarrow_{tgt} \sigma_m, T_{m+1}$$

Because of the induction hypothesis, we have

$$C, D, \sigma_k, A \Downarrow_{int} \sigma_{k+1}, T_{k+1} \text{ for all } 0 \le k \le m-1 \qquad C, D, \sigma_m, A \Downarrow_{int} \sigma_m, T_{m+1}.$$

If we define $\sigma_k = \sigma_m$, $T_k = T_{m+1}$ for all $m+1 \le k \le n-1$, then $C, D, \sigma_k, A \Downarrow_{int} \sigma_{k+1}, T_{k+1}$ for all $m \le k \le n-1$ because of the last premises. Grouping them with the third premise, we can construct $C, D, \sigma_k, A \Downarrow_{int} \sigma_{k+1}, T_{k+1}$ for all $0 \le k \le n-1$. Therefore, we can derive $(\textbf{loop\_fixpt\_noacc}(n)\ C), D, \sigma, A \Downarrow_{int} \sigma_n, T_n$, where $\sigma_n = \sigma_m$ and $T_n = T_{m+1}$.

We can prove the if direction likewise. Similarly to the only-if direction, the proof is by induction on the depth of the derivation of $\Downarrow_{int}$, and it is enough to prove only the **loop_fixpt_noacc** case. The premises from the original rule of **loop_fixpt_noacc** are as follow:

$$\sigma_0 = \sigma \qquad C, D, \sigma_k, A \Downarrow_{int} \sigma_{k+1}, T_{k+1} \text{ for all } k \in \{0, \ldots, n-1\},$$

where the inferred relation is as follows:

$$(\textbf{loop\_fixpt\_noacc}(n)\ C), D, \sigma, A \Downarrow_{int} \sigma_n, T_n$$

From the induction hypothesis, we have

$$C, D, \sigma_k, A \Downarrow_{tgt} \sigma_{k+1}, T_{k+1} \text{ for all } k \in \{0, \ldots, n-1\}$$

If $\sigma_k \neq \sigma_{k+1}$ for all $0 \le k < n-1$, we can use the second rule of **loop_fixpt_noacc** in $\Downarrow_{tgt}$ and obtain the following derivation: $(\textbf{loop\_fixpt\_noacc}(n)\ C), D, \sigma, A \Downarrow_{tgt} \sigma_n, T_n$.
Otherwise, $\sigma_k = \sigma_{k+1}$ for some $0 \le k < n-1$. We let m be the first such k. Then, we can use the first rule of **loop_fixpt_noacc** in $\Downarrow_{tgt}$ and obtain the following derivation:

$$(\textbf{loop\_fixpt\_noacc}(n)\ C), D, \sigma, A \Downarrow_{tgt} \sigma_m, T_{m+1}.$$

From the determinism of $\Downarrow_{int}$, we deduce $\sigma_k = \sigma_m$ for any $m \le k \le n$ and $T_k = T_{m+1}$ for any $m+1 \le k \le n$. Therefore, $\sigma_m = \sigma_n$, $T_{m+1} = T_n$. □

A.2 Couplings, Induced Relations, and Their Properties

One of the key tools in our proof is the notion of coupling, which we define below.

Definition 1 (Coupling). *A binary relation R on Index is a* **coupling** *if the following properties hold:*

(1) R is a relation of a partial bijection, that is, for all $i_0, i_1, i \in Index$,

$$i_0\ [R]\ i \ \wedge\ i_1\ [R]\ i \implies i_0 = i_1 \quad \textit{and} \quad i\ [R]\ i_0 \ \wedge\ i\ [R]\ i_1 \implies i_0 = i_1.$$

(2) For all $i_0, i_1 \in Index$ and $\alpha \in \mathrm{dom}(i_0) \cap \mathrm{dom}(i_1)$,

$$i_0\ [R]\ i_1 \implies i_0(\alpha) = i_1(\alpha).$$

Here, we write $i_0\ [R]\ i_1$ to mean $(i_0, i_1) \in R$.

A coupling R induces four relations. The first defines a relation on states, and two of the others define relations on tensors, while the last specifies a relation on sets of indices:

$$\begin{aligned}
\mathit{State}_{tgt}(R) &\subseteq \mathit{State}_{tgt} \times \mathit{State}_{tgt},\\
\sigma_0\ [\mathit{State}_{tgt}(R)]\ \sigma_1 &\iff \text{for all variables } x^{\tau^\dagger} \text{ and } i_0, i_1 \text{ with } i_0\ [R]\ i_1,\\
&\qquad (i)\ i_0 \in \mathsf{dom}(\sigma_0(x^{\tau^\dagger}))\uparrow \iff i_1 \in \mathsf{dom}(\sigma_1(x^{\tau^\dagger}))\uparrow,\\
&\qquad (ii)\ i_0 \in \mathsf{dom}(\sigma_0(x^{\tau^\dagger}))\uparrow \implies \mathit{extend}(\sigma_0(x^{\tau^\dagger}))(i_0) = \mathit{extend}(\sigma_1(x^{\tau^\dagger}))(i_1),
\end{aligned}$$

$$\begin{aligned}
\mathit{Tensor}_{[\![\tau]\!]}(R) &\subseteq \mathit{Tensor}_{[\![\tau]\!]} \times \mathit{Tensor}_{[\![\tau]\!]},\\
T_0\,[\mathit{Tensor}_{[\![\tau]\!]}(R)]\,T_1 &\iff \text{for all } i_0, i_1 \text{ with } i_0\ [R]\ i_1,\\
&\qquad (i)\ i_0 \in \mathsf{dom}(T_0) \iff i_1 \in \mathsf{dom}(T_1),\\
&\qquad (ii)\ i_0 \in \mathsf{dom}(T_0) \implies T_0(i_0) = T_1(i_1),
\end{aligned}$$

$$\begin{aligned}
\mathit{Tensor}^{\dagger}_{[\![\tau]\!]}(R) &\subseteq \mathit{Tensor}_{[\![\tau]\!]} \times \mathit{Tensor}_{[\![\tau]\!]},\\
T_0\,[\mathit{Tensor}^{\dagger}_{[\![\tau]\!]}(R)]\,T_1 &\iff \text{for all } i_0, i_1 \text{ with } i_0\ [R]\ i_1,\\
&\qquad (i)\ i_0 \in \mathsf{dom}(T_0)\uparrow \iff i_1 \in \mathsf{dom}(T_1)\uparrow,\\
&\qquad (ii)\ i_0 \in \mathsf{dom}(T_0)\uparrow \implies \mathit{extend}(T_0)(i_0) = \mathit{extend}(T_1)(i_1),
\end{aligned}$$

$$\begin{aligned}
\mathcal{P}(R) &\subseteq \mathcal{P}(\mathit{Index}) \times \mathcal{P}(\mathit{Index}),\\
L_0\ [\mathcal{P}(R)]\ L_1 &\iff \text{for all } i_0, i_1 \text{ with } i_0\ [R]\ i_1,\\
&\qquad (i)\ i_0 \in L_0\uparrow \iff i_1 \in L_1\uparrow,\\
&\qquad (ii)\ i_0 \in L_0\uparrow \implies \max(L_0 \cap i_0\downarrow)\ [R]\ \max(L_1 \cap i_1\downarrow).
\end{aligned}$$

One way to get used to these relations is to understand their common root: all the four relations are derived from the same general construction, which we describe next. Consider a set V and a relation S_R on V (i.e., $S_R \subseteq V \times V$) that may depend on R. The general construction is the relation $[R \rightharpoonup S_R]$ on partial functions of type $\mathit{Index} \rightharpoonup V$ that is defined as follows: for all $f_0, f_1 : \mathit{Index} \rightharpoonup V$,

$$\begin{aligned}
f_0\ [R \rightharpoonup S_R]\ f_1 &\iff \text{for all } i_0, i_1 \text{ with } i_0\ [R]\ i_1,\\
&\qquad (i)\ i_0 \in \mathsf{dom}(f_0) \iff i_1 \in \mathsf{dom}(f_1),\\
&\qquad (ii)\ i_0 \in \mathsf{dom}(f_0) \implies f_0(i_0)\ [S_R]\ f_1(i_1).
\end{aligned}$$

For a set W, let Δ_W be the equality relation on W:

$$\Delta_W = \{(w, w) : w \in W\},$$

which is also known as the diagonal relation on W. Then, the definitions of the four relations are equivalent to the following constructions:

$$\begin{aligned}
\sigma_0\ [\mathit{State}_{tgt}(R)]\ \sigma_1 &\iff \mathit{extend}(\sigma_0(x^{\tau^\dagger}))\ [R \rightharpoonup \Delta_{[\![\tau]\!]}]\ \mathit{extend}(\sigma_1(x^{\tau^\dagger})) \ \text{ for all } x^{\tau^\dagger},\\
T_0\ [\mathit{Tensor}_{[\![\tau]\!]}(R)]\ T_1 &\iff T_0\ [R \rightharpoonup \Delta_{[\![\tau]\!]}]\ T_1,\\
T_0\ [\mathit{Tensor}^{\dagger}_{[\![\tau]\!]}(R)]\ T_1 &\iff \mathit{extend}(T_0)\ [R \rightharpoonup \Delta_{[\![\tau]\!]}]\ \mathit{extend}(T_1),\\
L_0\ [\mathcal{P}(R)]\ L_1 &\iff \mathit{extend}(\mathrm{id}|_{L_0})\ [R \rightharpoonup R]\ \mathit{extend}(\mathrm{id}|_{L_1}),
\end{aligned}$$

where $\mathrm{id}|_L$ is the identity function over *Index* restricted to $L \subseteq \mathit{Index}$.

Before proceeding further, we re-state and prove Lemma 1 which will be used in this section.

Lemma 1. *Let σ be a state, $x^{\tau\dagger}$, $y^{\tau'\dagger}$ be variables, T be a tensor in $\mathit{Tensor}_{[\![\tau]\!]}$, and ρ be a finite partial injection on Index. Then, we have the following equations:*

$$\mathit{extend}(\sigma[x^{\tau\dagger} : T](y^{\tau'\dagger})) = \begin{cases} \mathit{extend}(\sigma(x^{\tau\dagger}))[\mathit{extend}(T)] & \text{if } y^{\tau'\dagger} \equiv x^{\tau\dagger}, \\ \mathit{extend}(\sigma(y^{\tau'\dagger})) & \text{otherwise,} \end{cases}$$

$$\mathit{extend}(\sigma\langle\rho\rangle(y^{\tau'\dagger})) = \mathit{extend}(\sigma(y^{\tau'\dagger}))[\mathit{extend}(T_{y,\rho})],$$

where $T_{y,\rho}(i) = \sigma(y^{\tau'\dagger})(\rho^{-1}(i))$ if $i \in \mathsf{image}(\rho) \cap \rho(\mathsf{dom}(\sigma(y^{\tau'\dagger})))$, and it is undefined otherwise.

Proof. We first assume $y^{\tau'\dagger} \equiv x^{\tau\dagger}$ and $i \in \mathsf{dom}(T)\uparrow$. We have to show

$$\mathit{extend}(\sigma[x^{\tau\dagger} : T](y^{\tau'\dagger}))(i) = \mathit{extend}(T)(i). \tag{7}$$

By the definition of the update operator,

$$L = \mathsf{dom}(\sigma[x^{\tau\dagger} : T](y^{\tau'\dagger})) = (\mathsf{dom}(\sigma(x^{\tau\dagger})) \setminus \mathsf{dom}(T)\uparrow) \cup \mathsf{dom}(T).$$

We claim that

$$\max(L \cap i\downarrow) = \max(\mathsf{dom}(T) \cap i\downarrow).$$

To see this, let $i' = \max(\mathsf{dom}(T) \cap i\downarrow)$. Then, $i' \in L \cap i\downarrow$. Suppose for contradiction that there exists $i'' \in L \cap i\downarrow$ such that $i'' \sqsupseteq i'$ but $i'' \neq i'$. Then, $i'' \in \mathsf{dom}(T)\uparrow$, which implies $i'' \in \mathsf{dom}(T)$ since $i'' \in L$. However, this contradicts the fact that i' is the maximum of $\mathsf{dom}(T) \cap i\downarrow$, since i'' is strictly greater than i' while being contained in $\mathsf{dom}(T) \cap i\downarrow$. Thus, $i' = \max(L \cap i\downarrow)$, which proves the claim. Now, to see Eq. (7), we have

$$\begin{aligned} \mathit{extend}(\sigma[x^{\tau\dagger} : T](y^{\tau'\dagger}))(i) &= \sigma[x^{\tau\dagger} : T](y^{\tau'\dagger})(\max(L \cap i\downarrow)) \\ &= \sigma[x^{\tau\dagger} : T](y^{\tau'\dagger})(\max(\mathsf{dom}(T) \cap i\downarrow)) \\ &= T(\max(\mathsf{dom}(T) \cap i\downarrow)) \\ &= \mathit{extend}(T)(i). \end{aligned}$$

For the other case when $y^{\tau'\dagger} \not\equiv x^{\tau\dagger}$ or $i \notin \mathsf{dom}(T)\uparrow$, we have to show

$$\mathit{extend}(\sigma[x^{\tau\dagger} : T](y^{\tau'\dagger}))(i) = \mathit{extend}(\sigma(y^{\tau'\dagger}))(i). \tag{8}$$

We claim that

$$i \notin \mathsf{dom}(T)\uparrow \implies \max(L \cap i\downarrow) \notin \mathsf{dom}(T)\uparrow.$$

Suppose for contradiction that $\max(L \cap i\downarrow) \in \mathsf{dom}(T)\uparrow$. Then, $\max(L \cap i\downarrow) \in \mathsf{dom}(T)$ since it belongs to L. However, this cannot happen since $\max(L \cap i\downarrow)$ is also contained in $i\downarrow$, and $\mathsf{dom}(T) \cap i\downarrow = \emptyset$ by the assumption that $i \notin \mathsf{dom}(T)\uparrow$. Thus, the claim holds. Now, we can show Eq. (8) as follows:

$$\begin{aligned} \mathit{extend}(\sigma[x^{\tau\dagger} : T](y^{\tau'\dagger}))(i) &= \sigma[x^{\tau\dagger} : T](y^{\tau'\dagger})(\max(L \cap i\downarrow)) \\ &= \sigma(y^{\tau'\dagger})(\max(L \cap i\downarrow)) \\ &= \mathit{extend}(\sigma(y^{\tau'\dagger}))(i), \end{aligned}$$

where the second equality follows from that $y^{\tau'\dagger} \not\equiv x^{\tau\dagger}$ or $\max(L \cap i\downarrow) \notin \mathsf{dom}(T)\uparrow$. The second result in the lemma directly follows from that

$$\sigma\langle\rho\rangle(y^{\tau'\dagger}) = \sigma[y^{\tau'\dagger} : T_{y,\rho}](y^{\tau'\dagger})$$

for every variable $y^{\tau'\dagger}$. □

Also, we note a few basic facts about these induced relations.

Lemma 2. *Let R be a coupling, and L_0, L_1 be sets of indices. If $L_0\ [\mathcal{P}(R)]\ L_1$, then we have the following equivalence for all $i_0, i_1 \in \mathit{Index}$ with $i_0\ [R]\ i_1$:*

$$i_0 \in L_0 \iff i_1 \in L_1.$$

Proof. Let R be a coupling, L_0, L_1 be sets of indices, and i_0, i_1 be indices such that

$$L_0\ [\mathcal{P}(R)]\ L_1 \quad \text{and} \quad i_0\ [R]\ i_1.$$

We will show that $i_0 \in L_0$ implies $i_1 \in L_1$; the other implication can be proved similarly. Assume that $i_0 \in L_0$. Then, $i_0 \in L_0{\uparrow}$, which implies that $i_1 \in L_1{\uparrow}$. Furthermore,

$$\max(L_0 \cap i_0{\downarrow})\ [R]\ \max(L_1 \cap i_1{\downarrow}).$$

But $i_0 = \max(L_0 \cap i_0{\downarrow})$ since $i_0 \in L_0$. As a result, $i_1 = \max(L_1 \cap i_1{\downarrow})$ since $i_0\ [R]\ i_1$ and R is a partial bijection, as R is a coupling. Thus, $i_1 \in L_1$. □

Corollary 1. *Let R be a coupling, and L_0, L_1 be finite sets of indices. Then,*

$$L_0\ [\mathcal{P}(R)]\ L_1 \implies T^z_{L_0}\ [\mathit{Tensor}_{\mathbb{R}}(R)]\ T^z_{L_1}.$$

The following lemma asserts that if $\mathrm{dom}(f_0)\ [\mathcal{P}(R)]\ \mathrm{dom}(f_1)$, then *extend* preserves the $[R \rightharpoonup \Delta_V]$ relation between f_0 and f_1.

Lemma 3. *Let R be a coupling, and f_0, f_1 be partial functions from Index to a set V. Then, if $\mathrm{dom}(f_0)\ [\mathcal{P}(R)]\ \mathrm{dom}(f_1)$, we have*

$$f_0\ [R \rightharpoonup \Delta_V]\ f_1 \implies \mathit{extend}(f_0)\ [R \rightharpoonup \Delta_V]\ \mathit{extend}(f_1).$$

Proof. Consider arbitrary indices i_0, i_1 with $i_0\ [R]\ i_1$. First, $\mathrm{dom}(f_0)\ [\mathcal{P}(R)]\ \mathrm{dom}(f_1)$ directly implies that

$$i_0 \in \mathrm{dom}(\mathit{extend}(f_0)) = \mathrm{dom}(f_0){\uparrow} \iff i_1 \in \mathrm{dom}(\mathit{extend}(f_1)) = \mathrm{dom}(f_1){\uparrow}.$$

Next, we further assume that $i_0 \in \mathrm{dom}(\mathit{extend}(f_0))$ and $i_1 \in \mathrm{dom}(\mathit{extend}(f_1))$, and show that

$$f_0(\max(\mathrm{dom}(f_0) \cap i_0{\downarrow})) = f_1(\max(\mathrm{dom}(f_1) \cap i_1{\downarrow})),$$

which gives the desired $\mathit{extend}(f_0)(i_0) = \mathit{extend}(f_1)(i_1)$. This directly follows from that

$$f_0\ [R \rightharpoonup \Delta_V]\ f_1 \quad \text{and} \quad \max(\mathrm{dom}(f_0) \cap i_0{\downarrow})\ [R]\ \max(\mathrm{dom}(f_1) \cap i_1{\downarrow}).$$

□

Corollary 2. *If τ is either* int *or* real, *and $\mathrm{dom}(T_0)\ [\mathcal{P}(R)]\ \mathrm{dom}(T_1)$ for $T_0, T_1 \in \mathit{Tensor}_{[\![\tau]\!]}$, we have*

$$T_0\ [\mathit{Tensor}_{[\![\tau]\!]}(R)]\ T_1 \implies T_0\ [\mathit{Tensor}^{\dagger}_{[\![\tau]\!]}(R)]\ T_1.$$

Lemma 4. *For all real tensors $T_0, T_1, T_0', T_1' \in \mathit{Tensor}_{\mathbb{R}}$ and all couplings R, we have*

$$T_0\ [\mathit{Tensor}_{\mathbb{R}}(R)]\ T_1 \wedge T_0'\ [\mathit{Tensor}_{\mathbb{R}}(R)]\ T_1' \implies (T_0 \oplus T_0')\ [\mathit{Tensor}_{\mathbb{R}}(R)]\ (T_1 \oplus T_1').$$

Proof. Pick $i_0, i_1 \in \mathit{Index}$ such that $i_0\ [R]\ i_1$. Then, the first condition in the definition of $\mathit{Tensor}_{\mathbb{R}}(R)$ holds as shown below:

$$\begin{aligned} i_0 \in \mathrm{dom}(T_0 \oplus T_0') &\iff i_0 \in \mathrm{dom}(T_0) \vee i_0 \in \mathrm{dom}(T_0') \\ &\iff i_1 \in \mathrm{dom}(T_1) \vee i_1 \in \mathrm{dom}(T_1') \\ &\iff i_1 \in \mathrm{dom}(T_1 \oplus T_1'), \end{aligned}$$

where the second equivalence follows from the assumption that

$$T_0\,[\mathit{Tensor}_{\mathbb{R}}(R)]\,T_1 \quad \text{and} \quad T_0'\,[\mathit{Tensor}_{\mathbb{R}}(R)]\,T_1'.$$

To prove the other condition in the definition of $\mathit{Tensor}_{\mathbb{R}}(R)$, assume that $i_0 \in \mathsf{dom}(T_0 \oplus T_0')$. If $i_0 \in \mathsf{dom}(T_0) \cap \mathsf{dom}(T_0')$, then

$$i_1 \in \mathsf{dom}(T_1) \cap \mathsf{dom}(T_1'), \quad T_0(i_0) = T_1(i_1) \quad \text{and} \quad T_0'(i_0) = T_1'(i_1),$$

and so

$$(T_0 \oplus T_0')(i_0) = T_0(i_0) + T_0'(i_0) = T_1(i_1) + T_1'(i_1) = (T_1 \oplus T_1')(i_1),$$

as desired. If $i_0 \in \mathsf{dom}(T_0) \setminus \mathsf{dom}(T_0')$, then $i_1 \in \mathsf{dom}(T_1) \setminus \mathsf{dom}(T_1')$ and $T_0(i_0) = T_1(i_1)$, and these facts imply the desired equality:

$$(T_0 \oplus T_0')(i_0) = T_0(i_0) = T_1(i_1) = (T_1 \oplus T_1')(i_1).$$

The remaining case $i_0 \in \mathsf{dom}(T_0') \setminus \mathsf{dom}(T_0)$ can be handled similarly to the previous case. □

Lemma 5. *Let R be a coupling, and $x^{\tau\dagger}$ be a variable. Consider states $\sigma_0, \sigma_1 \in \mathit{State}_{tgt}$ and tensors $T_0, T_1 \in \mathit{Tensor}_{[\![\tau]\!]}$ such that*

$$\sigma_0\,[\mathit{State}_{tgt}(R)]\,\sigma_1 \quad \textit{and} \quad T_0\,[\mathit{Tensor}^{\dagger}_{[\![\tau]\!]}(R)]\,T_1.$$

Then,

$$\sigma_0[x^{\tau\dagger} : T_0]\,[\mathit{State}_{tgt}(R)]\,\sigma_1[x^{\tau\dagger} : T_1].$$

Proof. Let $x^{\tau\dagger}$ be a variable. Consider a coupling R, states σ_0, σ_1, and tensors T_0, T_1, such that

$$\sigma_0\,[\mathit{State}_{tgt}(R)]\,\sigma_1 \quad \text{and} \quad T_0\,[\mathit{Tensor}^{\dagger}_{[\![\tau]\!]}(R)]\,T_1.$$

Pick an arbitrary variable $y^{\tau'\dagger}$ and arbitrary indices i_0, i_1 such that $i_0\,[R]\,i_1$. Then, we have

$$\begin{aligned}
\mathit{extend}(\sigma_0[x^{\tau\dagger} : T_0](y^{\tau'\dagger}))(i_0) &= \begin{cases} \mathit{extend}(\sigma_0(x^{\tau\dagger}))[\mathit{extend}(T_0)](i_0) & \text{if } y^{\tau'\dagger} \equiv x^{\tau\dagger}, \\ \mathit{extend}(\sigma_0(y^{\tau'\dagger}))(i_0) & \text{otherwise,} \end{cases} \\
&= \begin{cases} \mathit{extend}(T_0)(i_0) & \text{if } y^{\tau'\dagger} \equiv x^{\tau\dagger} \text{ and } i_0 \in \mathsf{dom}(T_0)\uparrow, \\ \mathit{extend}(\sigma_0(x^{\tau\dagger}))(i_0) & \text{if } y^{\tau'\dagger} \equiv x^{\tau\dagger} \text{ and } i_0 \notin \mathsf{dom}(T_0)\uparrow, \\ \mathit{extend}(\sigma_0(y^{\tau'\dagger}))(i_0) & \text{otherwise,} \end{cases} \\
&= \begin{cases} \mathit{extend}(T_1)(i_1) & \text{if } y^{\tau'\dagger} \equiv x^{\tau\dagger} \text{ and } i_1 \in \mathsf{dom}(T_1)\uparrow, \\ \mathit{extend}(\sigma_1(x^{\tau\dagger}))(i_1) & \text{if } y^{\tau'\dagger} \equiv x^{\tau\dagger} \text{ and } i_1 \notin \mathsf{dom}(T_1)\uparrow, \\ \mathit{extend}(\sigma_1(y^{\tau'\dagger}))(i_1) & \text{otherwise,} \end{cases} \\
&= \begin{cases} \mathit{extend}(\sigma_1(x^{\tau\dagger}))[\mathit{extend}(T_1)](i_1) & \text{if } y^{\tau'\dagger} \equiv x^{\tau\dagger}, \\ \mathit{extend}(\sigma_1(y^{\tau'\dagger}))(i_1) & \text{otherwise,} \end{cases} \\
&= \mathit{extend}(\sigma_1[x^{\tau\dagger} : T_1](y^{\tau'\dagger}))(i_1).
\end{aligned}$$

The first and the last equalities follow from Lemma 1. The third equality holds because $i_0\,[R]\,i_1$ and $\sigma_0\,[\mathit{State}_{tgt}(R)]\,\sigma_1$ and $T_0\,[\mathit{Tensor}^{\dagger}_{[\![\tau]\!]}(R)]\,T_1$. □

When $R = \Delta_L$ for a subset L of *Index*, we write $=_L$ for $\mathit{State}_{tgt}(R)$. We spell out the definition of $=_L$ explicitly below since it is used frequently:

$$\sigma_0 =_L \sigma_1 \iff \mathit{extend}(\sigma_0(x^{\tau\dagger}))(i) = \mathit{extend}(\sigma_1(x^{\tau\dagger}))(i) \text{ for all variables } x^{\tau\dagger} \text{ and indices } i \in L.$$

Intuitively, it says that the *extend* functions represented by $\sigma_0(x^{\tau\dagger})$ and $\sigma_1(x^{\tau\dagger})$ coincide on L.

Corollary 3 (Effects of Variable Writes). *For all states $\sigma \in State_{tgt}$, variables $x^{\tau\dagger}$, and tensors $T \in Tensor_{[\![\tau]\!]}$ such that $L = \mathsf{dom}(T)$, we have*

$$\sigma[x^{\tau\dagger} : T] =_{(L\uparrow)^c} \sigma.$$

Proof. It follows from Lemma 5 with the following facts:

$$\sigma =_{(L\uparrow)^c} \sigma \quad \text{and} \quad T\ [Tensor^{\dagger}_{[\![\tau]\!]}(\Delta_{(L\uparrow)^c})]\ T_{\emptyset} \quad \text{and} \quad \sigma[x^{\tau\dagger} : T_{\emptyset}] = \sigma.$$

□

Lemma 6 (Updated Map Entries: Commands). *Let C be a command in the target language, $D \in RDB$, and $\sigma, \sigma' \in State_{tgt}$. Suppose that $C, D, \sigma, A \Downarrow_{int} \sigma', T$. Then,*

$$\sigma =_{(A\uparrow)^c} \sigma' \qquad \textit{and} \qquad \mathsf{dom}(T) = A.$$

Intuitively, the lemma expresses that if an entry of the map stored in a variable is updated during execution, its index should be in the upward closure of the given A for the execution, and also that the tensor recording the outcome of scoring during execution should have domain A.

Proof. We prove the lemma by induction on the structure of C. We do case analysis on C, and prove that the two conditions of the lemma hold for each case.

- **Cases of $C \equiv (x^{\mathsf{real}\dagger} := \mathbf{fetch}(I))$, $C \equiv (x^{\mathsf{int}\dagger} := \mathbf{lookup\_index}(\alpha))$, $C \equiv (x^{\mathsf{int}\dagger} := Z)$, and $C \equiv (x^{\mathsf{real}\dagger} := E)$:**
  We have $\mathsf{dom}(T) = \mathsf{dom}(T^z_A) = A$. Thus, the second condition of the lemma holds. For the other condition, we note that by the rules for these commands in the semantics, $\sigma' = \sigma[x^{\mathsf{real}\dagger} : T']$ or $\sigma' = \sigma[x^{\mathsf{int}\dagger} : T']$ for some tensor T' with $\mathsf{dom}(T') = A$. Thus, by Corollary 3, we have the desired $\sigma =_{(A\uparrow)^c} \sigma'$.
- **Cases of $C \equiv (\mathbf{score}(E))$ and $C \equiv (\mathbf{skip})$:**
  In these two cases, $\mathsf{dom}(T) = A$, and so the second condition of the lemma holds. The other condition also holds because the input state σ is not modified in these cases.
- **Case of $C \equiv (C_1; C_2)$:**
  The second condition follows from the induction hypothesis on C_1 and C_2, and the fact that $\mathsf{dom}(T_1 \oplus T_2) = \mathsf{dom}(T_1) \cup \mathsf{dom}(T_2)$ for any $T_1, T_2 \in Tensor_{\mathbb{R}}$. The other condition holds because of the induction hypothesis on C_1 and C_2 and the transitivity of the $=_{(A\uparrow)^c}$ relation.
- **Case of $C \equiv (\mathbf{ifz}\ Z\ C_1\ C_2)$:**
  Let $A_1 = \{i \in A : n_i = 0\}$ and $A_2 = \{i \in A : n_i \neq 0\}$ where $n_i = [\![Z]\!](\sigma, i)$ for all $i \in A$. By the induction hypothesis on C_1 and C_2, we have

  $$C_1, D, \sigma, A_1 \Downarrow_{int} \sigma_1, T_1, \qquad \mathsf{dom}(T_1) = A_1, \qquad \sigma =_{(A_1\uparrow)^c} \sigma_1,$$
  $$C_2, D, \sigma_1, A_2 \Downarrow_{int} \sigma_2, T_2, \qquad \mathsf{dom}(T_2) = A_2, \qquad \sigma_1 =_{(A_2\uparrow^c)} \sigma_2,$$

  for $\sigma' = \sigma_2$ and $T = T_1 \oplus T_2$. The second condition of the lemma follows from these relationships as shown below:

  $$\mathsf{dom}(T) = \mathsf{dom}(T_1 \oplus T_2) = \mathsf{dom}(T_1) \cup \mathsf{dom}(T_2) = A_1 \cup A_2 = A.$$

  The other condition also holds because $(A\uparrow)^c \subseteq (A_1\uparrow)^c$ and $(A\uparrow)^c \subseteq (A_2\uparrow)^c$ imply

  $$\sigma =_{(A\uparrow)^c} \sigma_1 =_{(A\uparrow)^c} \sigma_2 = \sigma',$$

  and the relation $=_{(A\uparrow)^c}$ is transitive.

- **Cases of** $C \equiv (\textbf{for } x^{\mathsf{int}^\dagger} \textbf{ in range}(n) \textbf{ do } C')$**:**
  C in this case can be expressed as a sequence of $2n$ commands. By repeating our proofs for assignment and sequential composition and using the induction hypothesis on C', we can prove that the lemma holds in this case.
- **Case of** $C \equiv (\textbf{extend\_index}(\alpha, n)\ C')$**:**
  Let

$$A_0 = \{i +\!\!+ [(\alpha, k)] : i \in A,\ k \in \{0, \ldots, n-1\}\},$$
$$\rho_0 = [i +\!\!+ [(\alpha, n-1)] \mapsto i : i \in A].$$

  By the semantics of the **extend_index** command, there exist $\sigma_0 \in \mathit{State}_{tgt}$ and $T_0 \in \mathit{Tensor}_{\mathbb{R}}$ such that

$$C', D, \sigma, A_0 \Downarrow_{int} \sigma_0, T_0, \quad \sigma' = \sigma_0\langle\rho_0\rangle, \quad T = \left[i \mapsto \sum_{k=0}^{n-1} T_0(i +\!\!+ [(\alpha, k)]) \ :\ i \in A \right].$$

  By the induction hypothesis on C',

$$\sigma =_{(A_0\uparrow)^c} \sigma_0 \quad \text{and} \quad \mathsf{dom}(T_0) = A_0.$$

  Note that the second relationship from above ensures that the rule for the **extend_index** command in the operational semantics never experiences the failure due to the undefinedness of the tensor T_0 at some $i +\!\!+ [(\alpha, k)]$, as we explained right after the definition of the operational semantics. The second condition required by the lemma holds because $\mathsf{dom}(T) = A$. To prove the other condition, we note that for all variables $x^{\tau^\dagger}$ and indices $j \in (A\uparrow)^c$,

$$\begin{aligned} \mathit{extend}(\sigma'(x^{\tau^\dagger}))(j) &= \mathit{extend}(\sigma_0\langle\rho_0\rangle(x^{\tau^\dagger}))(j) \\ &= \mathit{extend}(\sigma_0(x^{\tau^\dagger}))(j) \\ &= \mathit{extend}(\sigma(x^{\tau^\dagger}))(j). \end{aligned}$$

  In the derivation, the second equality holds because $j \in (A\uparrow)^c = (\mathsf{image}(\rho_0)\uparrow)^c$, and the third equality holds because $\sigma =_{(A_0\uparrow)^c} \sigma_0$ and $(A\uparrow)^c \subseteq (A_0\uparrow)^c$.
- **Case of** $C \equiv \textbf{shift}(\alpha)$**:**
  In this case, $\mathsf{dom}(T) = \mathsf{dom}(T_A^z) = A$, as required by the second condition of the lemma. To prove the other condition, pick an arbitrary variable $x^{\tau^\dagger}$ and index $i \in (A\uparrow)^c$. Note that $\sigma' = \sigma\langle\rho\rangle$ for some ρ such that $\mathsf{image}(\rho) \subseteq A$.

$$\begin{aligned} \mathit{extend}(\sigma'(x^{\tau^\dagger}))(i) &= \mathit{extend}(\sigma\langle\rho\rangle(x^{\tau^\dagger}))(i) \\ &= \mathit{extend}(\sigma(x^{\tau^\dagger}))(i), \end{aligned}$$

  where the second equality holds because $i \in (A\uparrow)^c \subseteq (\mathsf{image}(\rho)\uparrow)^c$.
- **Case of** $C \equiv (\textbf{loop\_fixpt\_noacc}(n)\ C')$**:**
  By the semantics of the **loop_fixpt_noacc** command, there exist $\sigma_k \in \mathit{State}_{tgt}$ and $T_k \in \mathit{Tensor}_{\mathbb{R}}$ for $k = 0, \ldots, n-1$ such that

$$\sigma_0 = \sigma, \qquad \sigma_n = \sigma', \qquad T_n = T, \qquad C', D, \sigma_k, A \Downarrow_{int} \sigma_{k+1}, T_{k+1} \quad \text{for } k = 0, \ldots, n-1.$$

  By the induction hypothesis on C' and the transitivity of the $=_{(A\uparrow)^c}$ relation, we have $\sigma_k =_{(A\uparrow)^c} \sigma_{k+1}$ for $k = 0, \ldots, n-1$, and thus $\sigma =_{(A\uparrow)^c} \sigma'$ and $\mathsf{dom}(T) = A$.

□

Corollary 4 (Identity Property under Empty Set of Indices). *Let C be a command in the target language, $D \in RDB$, and $\sigma, \sigma' \in State_{tgt}$. Then, if $C, D, \sigma, \emptyset \Downarrow_{int} \sigma', T$, then*

$$\sigma =_{Index} \sigma' \quad \textit{and} \quad T = T_\emptyset$$

where $T_\emptyset$ is the empty tensor, i.e., a tensor with the empty domain.

Proof. This directly follows from Lemma 6 since

$$(\emptyset\uparrow)^c = Index \quad \text{and} \quad \mathrm{dom}(T) = \emptyset \iff T = T_\emptyset.$$

□

A.3 Embedding of Source-Language Commands into Target-Language Commands

The simple straightforward way to embed expressions and commands in the source language to those in the target language is simply to change the type of each variable so that a variable x of type τ in the source language becomes the variable with the same name but of the lifted type $\tau\dagger$ in the target language. We denote the outcomes of this embedding using the $-'$ notation. For instance, E' and C' denote the embeddings of an expression E and a command C in the source language.

In this subsection, we prove that this embedding preserves the behaviour of the commands. Specifically, we show that if the initial states of a source language command and its embedding in the target language are equivalent, then their final states and scores are also equivalent after evaluating the commands using the evaluation relations $\Downarrow$ and $\Downarrow_{int}$, respectively.

Consider the following two relations, the first between states of the source language and those of the target language, and the next between real numbers and real-valued tensors, both denoted with the same notation $\sim$:

$$\sim \subseteq State_{src} \times State_{tgt}, \qquad \sigma \sim \sigma' \iff \sigma(x^\tau) = extend(\sigma'(x^{\tau\dagger}))([]) \text{ for all variables } x^\tau \text{ in the source languages,}$$

$$\sim \subseteq \mathbb{R} \times Tensor_{\mathbb{R}}, \qquad r \sim T \iff r = \sum_{i \in \mathrm{dom}(T)} T(i).$$

The next proposition says that the embedding preserves the semantics of expressions.

Proposition 1 (Embedding of Expressions). *Let $\sigma \in State_{src}$ and $\sigma' \in State_{tgt}$ such that $\sigma \sim \sigma'$. Then, for all integer expressions Z, real expressions E, and index expressions I in the source language, if we let Z', E', and I' be the ones obtained from Z, E, and I by replacing all variables x^τ in them with $x^{\tau\dagger}$, then*

$$[\![Z']\!](\sigma', []) = [\![Z]\!](\sigma), \qquad [\![E']\!](\sigma', []) = [\![E]\!](\sigma), \qquad [\![I']\!](\sigma', []) = [\![I]\!](\sigma),$$

where the left-hand side of each equation uses the semantics of the target language, and the right-hand side uses the semantics of the source language.

Proof. The proof is by induction on the structures of E, Z, and I. In the proof, we use the primed version of an expression of any type to mean the one obtained from it by replacing all variables x^τ with $x^{\tau\dagger}$.

- **Case of $Z \equiv n$ and $E \equiv r$:**
  The desired equality follows immediately from the semantics of the source and target languages.
- **Case of $Z \equiv x^{\mathsf{int}}$ and $E \equiv x^{\mathsf{real}}$:**
  We derive the desired equality as follows:

$$[\![x^{\tau\dagger}]\!](\sigma', []) = extend(\sigma'(x^{\tau\dagger}))([]) = \sigma(x^\tau) = [\![x^\tau]\!](\sigma)$$

for $\tau \in \{\mathsf{int}, \mathsf{real}\}$. Here the second equality follows from the assumption that $\sigma \sim \sigma'$, and the other equalities from the semantics of the source and target languages.

- **Case of** $Z \equiv \mathrm{op}_i(Z_1, \ldots, Z_n, E_1, \ldots, E_m)$ **and** $E \equiv \mathrm{op}_r(Z_1, \ldots, Z_n, E_1, \ldots, E_m)$:
  For $\mathrm{op} \in \{\mathrm{op}_i, \mathrm{op}_r\}$, we show the desired equality as follows:

$$\begin{aligned}
[\![\mathrm{op}(Z'_1, \ldots, Z'_n, E'_1, \ldots, E'_m)]\!](\sigma', []) &= ([\![\mathrm{op}]\!]_a)([\![Z'_1]\!](\sigma', []), \ldots, [\![Z'_m]\!](\sigma', []), \\
&\qquad [\![E'_1]\!](\sigma', []), \ldots, [\![E'_n]\!](\sigma', [])) \\
&= ([\![\mathrm{op}]\!]_a)([\![Z_1]\!](\sigma), \ldots, [\![Z_m]\!](\sigma), \\
&\qquad [\![E_1]\!](\sigma), \ldots, [\![E_n]\!](\sigma)) \\
&= [\![\mathrm{op}(Z_1, \ldots, Z_n, E_1, \ldots, E_m)]\!](\sigma).
\end{aligned}$$

  The second and third equalities use the induction hypothesis, and the other equalities follow from the semantics of the source and target languages.

- **Case of** $I \equiv [(\alpha_1, Z_1); \ldots; (\alpha_n, Z_n)]$:
  We prove the goal equality as shown below:

$$\begin{aligned}
[\![[(\alpha_1, Z'_1); \ldots; (\alpha_n, Z'_n)]]\!](\sigma', []) &= [(\alpha_1, [\![Z'_1]\!](\sigma', [])); \ldots; (\alpha_n, [\![Z'_n]\!](\sigma', []))] \\
&= [(\alpha_1, [\![Z_1]\!](\sigma)); \ldots; (\alpha_n, [\![Z_n]\!](\sigma))] \\
&= [\![[(\alpha_1, Z_1); \ldots; (\alpha_n, Z_n)]]\!](\sigma).
\end{aligned}$$

  The first equality follows from the induction hypothesis and the semantics of the index expression in the target language. The second equality uses the induction hypothesis, and the last equality the semantics of the source language.

□

We next show that the embedding preserves the semantics of commands.

Lemma 7 (Preservation of Equivalence by Update)**.** *Let $\sigma \in State_{src}$ and $\sigma' \in State_{tgt}$. If $\sigma \sim \sigma'$, then for all variables x^τ in the source language and all $r \in \mathbb{R}$, we have*

$$\sigma[x^\tau \mapsto r] \sim \sigma'[x^{\tau\dagger} : T], \quad \text{for } T = [[] \mapsto r].$$

Proof. Let σ, σ', x^τ, r, and T be as in the lemma. Assume that $\sigma \sim \sigma'$. For all variables $y^{\tau'}$ in the source language, we have

$$\begin{aligned}
\sigma[x^\tau \mapsto r](y^{\tau'}) &= \begin{cases} r & \text{if } y^{\tau'} \equiv x^\tau \\ \sigma(y^{\tau'}) & \text{otherwise} \end{cases} \\
&= \begin{cases} \mathit{extend}(\sigma'[x^{\tau\dagger} : T](x^{\tau\dagger}))([]) & \text{if } y^{\tau'} \equiv x^\tau \\ \mathit{extend}(\sigma'(y^{\tau'\dagger}))([]) & \text{otherwise} \end{cases} \\
&= \mathit{extend}(\sigma'[x^{\tau\dagger} : T](y^{\tau'\dagger}))([]).
\end{aligned}$$

□

Proposition 2. *Let C be a command in the source language, and C' be the command in the target language obtained from C by replacing all variables x^τ with $x^{\tau\dagger}$. Then, for all $\sigma_0 \in State_{src}$ and $\sigma'_0 \in State_{tgt}$ such that $\sigma_0 \sim \sigma'_0$, we have the following two properties:*

(1) if $C, D, \sigma_0 \Downarrow \sigma_1, r$ for some $\sigma_1 \in State_{src}$ and $r \in \mathbb{R}$, then there exist $\sigma'_1 \in State_{tgt}$ and $T \in Tensor_{\mathbb{R}}$ such that

$$C', D, \sigma'_0, \{[]\} \Downarrow_{int} \sigma'_1, T, \qquad \sigma_1 \sim \sigma'_1, \qquad r \sim T,$$

(2) if $C', D, \sigma_0', \{[]\} \Downarrow_{int} \sigma_1', T$ *for some* $\sigma_1' \in State_{tgt}$ *and* $T \in Tensor_{\mathbb{R}}$*, then there exist* $\sigma_1 \in State_{src}$ *and* $r \in \mathbb{R}$ *such that*

$$C, D, \sigma_0 \Downarrow \sigma_1, r, \qquad \sigma_1 \sim \sigma_1', \qquad r \sim T.$$

Proof. Let $A = \{[]\} \in FAChain$. We prove the proposition by structural induction on C.

- **Case of** $C \equiv (x^{\text{real}} := \textbf{fetch}(I))$ **and** $C' \equiv (x^{\text{real}\dagger} := \textbf{fetch}(I'))$:

(1) Assume that $C, D, \sigma_0 \Downarrow \sigma_1, r$ for some σ_1 and r. By the semantics of the source language,

$$\sigma_1 = \sigma[x^{\text{real}} \mapsto D(\llbracket I \rrbracket(\sigma_0))], \qquad r = 0.0.$$

By Proposition 1, the assumption $\sigma_0 \sim \sigma_0'$ entails that $\llbracket I' \rrbracket(\sigma_0', []) = \llbracket I \rrbracket(\sigma_0)$. By the semantics of the target language,

$$C', D, \sigma_0', A \Downarrow_{int} \sigma_1', T, \qquad \sigma_1' = \sigma_0'[x^{\text{real}\dagger} : [[] \mapsto D(\llbracket I \rrbracket(\sigma_0))]], \qquad T = T_A^z.$$

Lemma 7 entails that $\sigma_1 \sim \sigma_1'$. Last, $r \sim T$ clearly holds.

(2) Assume that $C', D, \sigma_0', A \Downarrow_{int} \sigma_1', T$ for some σ_1' and T. By the semantics of the target language,

$$\sigma_1' = \sigma[x^{\text{real}\dagger} : [[] \mapsto D(\llbracket I' \rrbracket(\sigma', []))]], \qquad T = T_A^z.$$

By Proposition 1, $\llbracket I' \rrbracket(\sigma', []) = \llbracket I \rrbracket(\sigma)$. By the semantics of the source language,

$$C, D, \sigma \Downarrow \sigma_0', r, \qquad \sigma_0' = \sigma[x^{\text{real}} \mapsto D(\llbracket I \rrbracket(\sigma))], \qquad r = 0.0$$

Lemma 7 entails that $\sigma_1 \sim \sigma_1'$. Last, $r \sim T$ clearly holds.

- **Case of** $C \equiv (\textbf{score}(E))$ **and** $C' \equiv (\textbf{score}(E'))$:

(1) Assume that $C, D, \sigma_0 \Downarrow \sigma_1, r$ for some σ_1 and r. By the semantics of the source language,

$$\sigma_0 = \sigma_1 \qquad r = \llbracket E \rrbracket(\sigma_0).$$

By Proposition 1, $\llbracket E' \rrbracket(\sigma_0', []) = \llbracket E \rrbracket(\sigma_0) = r$. Let $\sigma_1' = \sigma_0'$ and $T = [[] \mapsto r]$. Then, we have

$$C', D, \sigma_0', A \Downarrow_{int} \sigma_1', T, \qquad \sigma_1 \sim \sigma_1', \qquad r \sim T.$$

(2) Assume that $C', D, \sigma_0', A \Downarrow_{int} \sigma_1', T$ for some σ_1' and T. By the semantics of the target language

$$\sigma_0' = \sigma_1', \qquad T = [[] \mapsto \llbracket E' \rrbracket(\sigma_0', [])].$$

By Proposition 1, $\llbracket E \rrbracket(\sigma_0) = \llbracket E' \rrbracket(\sigma_0', [])$. Define $\sigma_1 = \sigma_0$ and $r = \llbracket E \rrbracket(\sigma_0)$. Then, we have

$$C, D, \sigma_0 \Downarrow \sigma_1, r, \qquad \sigma_1 \sim \sigma_1', \qquad r \sim T.$$

- **Case of** $C \equiv (x^{\text{int}} := Z)$ **and** $C' \equiv (x^{\text{int}\dagger} := Z')$:

(1) Assume that $C, D, \sigma_0 \Downarrow \sigma_1, r$ for some σ_1 and r. By the semantics of the source language,

$$\sigma_1 = \sigma_0[x^{\text{int}} \mapsto \llbracket Z \rrbracket(\sigma_0)], \qquad r = 0.0.$$

By Proposition 1, $\llbracket Z' \rrbracket(\sigma_0', []) = \llbracket Z \rrbracket(\sigma_0)$. By the semantics of the target language,

$$C', D, \sigma_0', A \Downarrow_{int} \sigma_1', T, \qquad \sigma_1' = \sigma_0'[x^{\text{int}\dagger} : [[] \mapsto \llbracket Z' \rrbracket(\sigma_0', [])]], \qquad T = T_A^z.$$

Lemma 7 entails $\sigma_1 \sim \sigma_1'$, Last, $r \sim T$ is obvious.

(2) Assume that $C', D, \sigma_0', A \Downarrow_{int} \sigma_1', T$ for some σ_1' and T. By the semantics of the target language,

$$\sigma_1' = \sigma_0'[x^{\text{int}^\dagger} : [[] \mapsto [\![Z']\!](\sigma_0', [])]], \qquad T = T_A^z.$$

By Proposition 1, $[\![Z]\!](\sigma_0) = [\![Z']\!](\sigma_0', [])$. By the semantics of the source language,

$$C, D, \sigma_0 \Downarrow \sigma_1, r, \qquad \sigma_1 = \sigma[x^{\text{int}} \mapsto [\![Z]\!](\sigma_0)], \qquad r = 0.0.$$

Lemma 7 entails $\sigma_1 \sim \sigma_1'$. Last, $r \sim T$ is obvious.

- **Case of** $C \equiv (x^{\text{real}} := E)$ **and** $C' \equiv (x^{\text{real}^\dagger} := E')$:
  This case is similar to the previous one.

- **Case of** $C \equiv C' \equiv$ **skip**:
  In this case, by the semantics of the source language and that of the target language, we have

$$\textbf{skip}, D, \sigma_0 \Downarrow \sigma_0, 0.0, \qquad \textbf{skip}, D, \sigma_0', A \Downarrow_{int} \sigma_0', T_A^z.$$

  Thus, the claimed properties of the proposition hold.

- **Case of** $C \equiv (C_1; C_2)$ **and** $C' \equiv (C_1'; C_2')$:

(1) Assume that $C_1; C_2, D, \sigma_0 \Downarrow \sigma_1, r$ for some σ_1 and r. By the semantics of the source language, there exist $r_1, r_2 \in \mathbb{R}$ and a state σ_2 such that

$$C_1, D, \sigma_0 \Downarrow \sigma_2, r_2, \qquad C_2, D, \sigma_2 \Downarrow \sigma_1, r_1, \qquad r = r_2 + r_1.$$

By the induction hypothesis on C_1 and C_1', there exist σ_2' and T_2 such that

$$C_1', D, \sigma_0', A \Downarrow_{int} \sigma_2', T_2, \qquad \sigma_2 \sim \sigma_2', \qquad r_2 \sim T_2.$$

Similarly, by the induction hypothesis on C_2 and C_2', there exist σ_1' and T_1 such that

$$C_2', D, \sigma_2', A \Downarrow_{int} \sigma_1', T_1, \qquad \sigma_1 \sim \sigma_1', \qquad r_1 \sim T_1.$$

Letting $T = T_2 \oplus T_1$ gives us

$$C_1'; C_2', D, \sigma_0', A \Downarrow_{int} \sigma_1', T, \qquad \sigma_0 \sim \sigma_1', \qquad r = r_2 + r_1.$$

(2) Assume that $C_1'; C_2', D, \sigma_0', A \Downarrow_{int} \sigma_1', T$ for some σ_1' and T. By the semantics of the target language, there exist T_1, T_2 and a state σ_2' such that

$$C_1', D, \sigma_0', A \Downarrow_{int} \sigma_2', T_2, \qquad C_2', D, \sigma_2', A \Downarrow_{int} \sigma_1', T_1, \qquad \sigma_0' \sim \sigma_2', \qquad T = T_2 \oplus T_1.$$

By the induction hypothesis on C_1 and C_1', there exist σ_2 and r_2 such that

$$C_1, D, \sigma_0 \Downarrow \sigma_2, r_2, \qquad \sigma_2 \sim \sigma_2', \qquad r_2 \sim T_2.$$

Similarly, by the induction hypothesis on C_2 and C_2', there exist σ_1 and r_1 such that

$$C_2, D, \sigma_2 \Downarrow \sigma_1, r_1, \qquad \sigma_1 \sim \sigma_1', \qquad r_1 \sim T_1.$$

Letting $r = r_2 + r_1$ gives us

$$C_1; C_2, D, \sigma_0 \Downarrow \sigma_1, r, \qquad \sigma_1 \sim \sigma_1', \qquad r \sim T.$$

- **Case of** $C \equiv (\textbf{ifz}\ Z\ C_1\ C_2)$ **and** $C' \equiv (\textbf{ifz}\ Z'\ C_1'\ C_2')$:

(1) Assume that $(\textbf{ifz}\ Z\ C_1\ C_2), D, \sigma_0 \Downarrow \sigma_1, r$ for some σ_1 and r. There are two subcases depending on the value of $[\![Z]\!](\sigma_0)$:

– **Subcase of** $[\![Z]\!](\sigma_0) = 0$:
By the semantics of the source language, we have $C_1, D, \sigma_0 \Downarrow \sigma_1, r$. Since $\sigma_0 \sim \sigma_0'$, we have $[\![Z']\!](\sigma_0', []) = [\![Z]\!](\sigma_0) = 0$ by Proposition 1. Therefore, with the notations of the rule for **ifz** in the target language semantics, $A_1 = \{[]\}$ and $A_2 = \emptyset$. By applying the induction hypothesis to C_1 and C_1', we derive the existence of σ_2' and T such that

$$C_1', D, \sigma_0', A_1 \Downarrow_{int} \sigma_2', T, \qquad \sigma_1 \sim \sigma_2', \qquad r \sim T,$$

Thus, by applying Corollary 4 to C_2', there exists σ_1' such that

$$C_2', D, \sigma_2', A_2 \Downarrow_{int} \sigma_1', T_\emptyset, \qquad \sigma_2' =_{Index} \sigma_1'.$$

Therefore, we conclude $(\textbf{ifz}\ Z'\ C_1'\ C_2'), D, \sigma_0', A \Downarrow_{int} \sigma_1', T$ with $\sigma_1 \sim \sigma_1'$ and $r \sim T$.

– **Subcase of** $[\![Z]\!](\sigma_0) \neq 0$:
By the semantics of the source language, we have $C_2, D, \sigma_0 \Downarrow \sigma_1, r$. Since $\sigma_0 \sim \sigma_0'$, we have $[\![Z']\!](\sigma_0', []) = [\![Z]\!](\sigma_0) \neq 0$ by Proposition 1. Therefore, with the notations of the rule for **ifz** in the semantics of the target language, $A_1 = \emptyset$ and $A_2 = \{[]\}$. By applying Corollary 4 to C_1', there exists σ_2' such that

$$C_1', D, \sigma_0', A_1 \Downarrow_{int} \sigma_2', T_\emptyset, \qquad \sigma_0' =_{Index} \sigma_2'.$$

This implies $\sigma_0 \sim \sigma_2'$, so we can apply the induction hypothesis to C_2 and C_2' to derive the existence of σ_1' and T such that

$$C_2', D, \sigma_2', A_1 \Downarrow_{int} \sigma_1', T, \qquad \sigma_1 \sim \sigma_1', \qquad r \sim T.$$

Therefore, we conclude $(\textbf{ifz}\ Z'\ C_1'\ C_2'), D, \sigma_0', A \Downarrow_{int} \sigma_1', T$ with $\sigma_1 \sim \sigma_1'$ and $r \sim T$.

(2) Now assume that $(\textbf{ifz}\ Z'\ C_1'\ C_2'), D, \sigma_0', A \Downarrow_{int} \sigma_1', T$ for some σ_1' and T. As before, we distinguish two subcases, depending on the value of $[\![Z']\!](\sigma_0', [])$:

– **Subcase of** $[\![Z']\!](\sigma_0', []) = 0$:
By the semantics of the target language, there exist σ_2' and T_2, T_1 such that

$$C_1', D, \sigma_0', \{[]\} \Downarrow_{int} \sigma_2', T_2, \qquad C_2', D, \sigma_2', \emptyset \Downarrow_{int} \sigma_1', T_1. \qquad T = T_2 \oplus T_1,$$

Since $\sigma_0 \sim \sigma_0'$, we have $[\![Z]\!](\sigma_0', []) = [\![Z']\!](\sigma_0) = 0$ by Proposition 1. By the induction hypothesis on C_1 and C_1', there exist σ_1 and r such that

$$C_1, D, \sigma_0 \Downarrow \sigma_1, r, \qquad \sigma_1 \sim \sigma_2', \qquad r \sim T_2.$$

By the semantics of the source language, this in turn implies that $(\textbf{ifz}\ Z\ C_1\ C_2), D, \sigma_0 \Downarrow \sigma_1, r$. Corollary 4 can be applied to C_2', and doing so gives the following facts:

$$\sigma_2' =_{Index} \sigma_1', \qquad T_1 = T_\emptyset.$$

So, $T_2 = T$ and $r \sim T$. Also, $\sigma_1 \sim \sigma_1'$.

– **Subcase of** $[\![Z']\!](\sigma_0', []) \neq 0$:
By the semantics of the target language, there exist σ_2' and T_2, T_1 such that

$$C_1', D, \sigma_0', \emptyset \Downarrow_{int} \sigma_2', T_2, \qquad C_2', D, \sigma_2', \{[]\} \Downarrow_{int} \sigma_1', T_1, \qquad T = T_2 \oplus T_1.$$

Since $\sigma_0 \sim \sigma_0'$, we have $[\![Z]\!](\sigma_0) = [\![Z']\!](\sigma_0', []) \neq 0$ by Proposition 1. Thus, we can apply Corollary 4 to C_1', and derive that

$$\sigma_0' =_{Index} \sigma_2', \qquad \sigma_0 \sim \sigma_2', \qquad T_2 = T_\emptyset, \qquad T = T_1.$$

By the induction hypothesis applied to C_2 and C_2', there exist σ_0' and r such that

$$C_2, D, \sigma_0 \Downarrow \sigma_1, r, \qquad \sigma_1 \sim \sigma_1', \qquad r \sim T.$$

By the semantics of the source language, $(\textbf{ifz}\ Z\ C_1\ C_2), D, \sigma_0 \Downarrow \sigma_1, r$.

- **Case of** $C \equiv (\textbf{for}\ x^{\mathsf{int}}\ \textbf{in range}(n)\ \textbf{do}\ C_1)$ **and** $C' \equiv (\textbf{for}\ x^{\mathsf{int}^\dagger}\ \textbf{in range}(n)\ \textbf{do}\ C_1')$:
  We observe that C is semantically equivalent to $x^{\mathsf{int}} := 0; C_1; x^{\mathsf{int}} := 1; C_1; \ldots x^{\mathsf{int}} := n-1; C_1$ and that C' is semantically equivalent to $x^{\mathsf{int}^\dagger} := 0; C_1'; x^{\mathsf{int}^\dagger} := 1; C_1'; \ldots x^{\mathsf{int}^\dagger} := n-1; C_1'$. Therefore the proof follows from the induction hypothesis on C_1 and C_1', and on the above arguments for the integer-assignment and sequential-composition cases, it follows that these loop cases satisfy the claimed properties of the proposition.

□

A.4 Soundness of Transformation of For-loops in the Target Language

In this subsection, we prove the soundness of the optimisation of for-loops within the target language, but with the evaluation rules defined by $\Downarrow_{int}$. Our optimisation replaces each for-loop by a **loop_fixpt_noacc** loop such that the body of the **loop_fixpt_noacc** loop works on a larger set of indices than the original loop and also runs without accumulating the score maps for all but the last iteration. We will prove that this optimisation preserves the original loop's semantics.

Before delving into the soundness theorem of our optimisation, we first introduce the concept of L-states. Let L be a set of indices. A state σ is called L-**state** if for every variable $x^{\tau^\dagger}$,

$$\mathsf{dom}(\sigma(x^{\tau^\dagger})) \subseteq L{\downarrow}.$$

For all L-states σ, variables $x^{\tau^\dagger}$, and indices i, we have

$$\mathit{extend}(\sigma(x^{\tau^\dagger}))(i) = \mathit{extend}(\sigma(x^{\tau^\dagger}))(\max(L{\downarrow} \cap i{\downarrow})).$$

That is, the values of $\mathit{extend}(\sigma(x^{\tau^\dagger}))$ at indices in $L{\downarrow}$ fully determines the entire function $\mathit{extend}(\sigma(x^{\tau^\dagger}))$. The next lemma gives that the update of a variable preserves the property of being an L-state.

Lemma 8. *Let L be a set of indices. Then, for all states $\sigma \in \mathit{State}_{tgt}$, variables $x^{\tau^\dagger}$, and tensors $T \in \mathit{Tensor}_{[\![\tau]\!]}$, if σ is an L-state and $\mathsf{dom}(T) \subseteq L{\downarrow}$, then $\sigma[x^{\tau^\dagger} : T]$ is also an L-state.*

Proof. For every variable $y^{\tau'^\dagger}$, we have

$$\mathsf{dom}(\sigma[x^{\tau^\dagger} : T](y^{\tau'^\dagger})) \subseteq \mathsf{dom}(\sigma(y^{\tau'^\dagger})) \cup \mathsf{dom}(T) \subseteq L{\downarrow}.$$

The second inclusion holds since σ is an L-state and $\mathsf{dom}(T) \subseteq L{\downarrow}$. Thus, $\sigma[x^{\tau^\dagger} : T]$ is an L-state. □

The following lemma says that the execution under a finite antichain $A \in \mathit{FAChain}$ preserves the state being an L-state for every L with $A \subseteq L{\downarrow}$.

Lemma 9 (L-state Preservation)**.** *Let C be a command, D be an RDB, A be a finite antichain of indices, and L be a set of indices. For all states $\sigma, \sigma' \in \mathit{State}_{tgt}$ and tensors $T \in \mathit{Tensor}_{\mathbb{R}}$, if*

$$A \subseteq L{\downarrow}, \quad \sigma \text{ is an } L\text{-state}, \quad \text{and} \quad C, D, \sigma, A \Downarrow_{int} \sigma', T,$$

then σ' is also an L-state.

Proof. The proof is by induction on the structure of C.

- **Cases of** $C \equiv (x^{\mathsf{real}^\dagger} := \textbf{fetch}(I))$, $C \equiv (\textbf{score}(E))$, $C \equiv (x^{\mathsf{int}^\dagger} := Z)$, $C \equiv (x^{\mathsf{real}^\dagger} := E)$, $C \equiv (\textbf{skip})$, **and** $C \equiv (x^{\mathsf{int}^\dagger} := \textbf{lookup\_index}(\alpha))$**:**
  In these cases, σ' is σ or $\sigma[x^{\tau^\dagger} : T]$ for some T with $\mathsf{dom}(T) = A \subseteq L{\downarrow}$. Thus, σ' is also an L-state by Lemma 8.
- **Cases of** $C \equiv (C_1; C_2)$**:**
  The desired conclusion follows from the induction hypothesis on C_1 and C_2.

- **Case of** $C \equiv (\textbf{ifz}\ Z\ C_1\ C_2)$**:**
  Let $A_1 = \{i \in A : [\![Z]\!](\sigma, i) = 0\}$ and $A_2 = \{i \in A : [\![Z]\!](\sigma, i) \neq 0\}$. Since $A_1 \subseteq L{\downarrow}$ and $A_2 \subseteq L{\downarrow}$, the desired conclusion follows from the induction hypothesis on C_1 and C_2.

- **Cases of** $C \equiv (\textbf{for}\ x^{\mathsf{int}^\dagger}\ \textbf{in range}(n)\ \textbf{do}\ C')$**:**
  We remark that C is semantically equivalent to $x^{\mathsf{int}^\dagger} := 0; C'; x^{\mathsf{int}^\dagger} := 1; C'; \ldots x^{\mathsf{int}^\dagger} := n-1; C'$. Therefore, the result follows from the induction hypothesis for C' and the above arguments for integer-assignments and sequential-composition.

- **Case of** $C \equiv (\textbf{extend\_index}(\alpha, n)\ C')$**:**
  By the rule of the **extend_index** command, we have

$$C', D, \sigma, A_0 \Downarrow_{int} \sigma_0, T_0, \qquad A_0 = \{i +\!\!+ [(\alpha, k)] : i \in A, k \in \{0, \ldots, n-1\}\},$$
$$\sigma' = \sigma_0\langle\rho_0\rangle, \qquad \rho_0 = [i +\!\!+ [(\alpha, n-1)] \mapsto i : i \in A].$$

  Then, σ is an $(L \cup A_0)$-state, and by the induction hypothesis on C', σ_0 is also an $(L \cup A_0)$-state. For an arbitrary variable $x^{\tau^\dagger}$, define a tensor $T_{x,\rho_0} \in \mathit{Tensor}_{[\![\tau]\!]}$ by

$$T_{x,\rho_0}(i) = \begin{cases} \sigma_0(x^{\tau^\dagger})(\rho_0^{-1}(i)), & \text{if } i \in \mathsf{image}(\rho_0) \cap \rho_0(\mathsf{dom}(\sigma_0(x^{\tau^\dagger}))), \\ \text{undefined}, & \text{otherwise.} \end{cases}$$

  Then,

$$\begin{aligned} \mathsf{dom}(\sigma'(x^{\tau^\dagger})) &= \mathsf{dom}(\sigma_0\langle\rho_0\rangle(x^{\tau^\dagger})) \\ &= \mathsf{dom}(\sigma_0[x^{\tau^\dagger} : T_{x,\rho_0}](x^{\tau^\dagger})) \\ &= (\mathsf{dom}(\sigma_0(x^{\tau^\dagger})) \setminus \mathsf{dom}(T_{x,\rho_0}){\uparrow}) \cup \mathsf{dom}(T_{x,\rho_0}) \\ &\subseteq ((L \cup A_0){\downarrow} \setminus \mathsf{dom}(T_{x,\rho_0}){\uparrow}) \cup \mathsf{dom}(T_{x,\rho_0}) \\ &= ((L \cup A_0){\downarrow} \setminus A{\uparrow}) \cup A \\ &\subseteq (L{\downarrow} \cup (A_0{\downarrow} \setminus A{\uparrow})) \cup A \\ &\subseteq (L{\downarrow} \cup A{\downarrow}) \cup A = L{\downarrow}. \end{aligned}$$

  The first subset relationship holds because σ_0 is an $(L \cup A_0)$-state, the fifth equality is true since $\mathsf{dom}(T_{x^{\tau^\dagger}}) = \mathsf{image}(\rho_0) = A$, and the third subset relationship holds because $A_0{\downarrow} = A{\downarrow} \cup A_0$ and $A_0 \subseteq A{\uparrow}$.

- **Case of** $C \equiv \textbf{shift}(\alpha)$**:**
  In this case, $\sigma' = \sigma\langle\rho\rangle$ for some finite partial function ρ such that $\mathsf{image}(\rho) \subseteq A$. For an arbitrary variable $x^{\tau^\dagger}$, define a tensor $T_{x,\rho} \in \mathit{Tensor}_{[\![\tau]\!]}$ by

$$T_{x,\rho}(i) = \begin{cases} \sigma(x^{\tau^\dagger})(\rho^{-1}(i)), & \text{if } i \in \mathsf{image}(\rho) \cap \rho(\mathsf{dom}(\sigma(x^{\tau^\dagger}))), \\ \text{undefined}, & \text{otherwise.} \end{cases}$$

  Then,

$$\begin{aligned} \mathsf{dom}(\sigma'(x^{\tau^\dagger})) &= \mathsf{dom}(\sigma\langle\rho\rangle(x^{\tau^\dagger})) \\ &= \mathsf{dom}(\sigma[x^{\tau^\dagger} : T_{x,\rho}](x^{\tau^\dagger})) \\ &= (\mathsf{dom}(\sigma(x^{\tau^\dagger})) \setminus \mathsf{dom}(T_{x,\rho}){\uparrow}) \cup \mathsf{dom}(T_{x,\rho}) \\ &\subseteq L{\downarrow} \cup \mathsf{image}(\rho) \\ &\subseteq L{\downarrow} \cup A = L{\downarrow}. \end{aligned}$$

  We have just shown that σ' is an L-state, as required.

- **Case of** $C \equiv (\textbf{loop\_fixpt\_noacc}(n)\ C')$**:**
  By the semantics of the **loop_fixpt_noacc** command, there exist $\sigma_k \in \mathit{State}_{tgt}$ and $T_k \in \mathit{Tensor}_{\mathbb{R}}$ for $k = 0, \ldots, n-1$ such that

  $$\sigma_0 = \sigma, \qquad \sigma_n = \sigma', \qquad T_n = T, \qquad C', D, \sigma_k, A \Downarrow_{int} \sigma_{k+1}, T_{k+1} \quad \text{for } k = 0, \ldots, n-1.$$

  By the induction hypothesis on C', σ_{k+1} is an L-state for all $k = 0, \ldots, n-1$, and thus σ' is also an L-state.

□

The following proposition states the preservation of lifted integer, real, and index expressions under a coupling R and the induced relations.

Proposition 3. *Let R be a coupling, and Z, E, I be lifted integer, real, and index expressions in the target language. Then, for all states $\sigma_0, \sigma_1 \in \mathit{State}_{tgt}$ and indices $i_0, i_1 \in \mathit{Index}$, if*

$$\sigma_0\ [\mathit{State}_{tgt}(R)]\ \sigma_1 \quad \textit{and} \quad i_0\ [R]\ i_1,$$

then

$$[\![Z]\!](\sigma_0, i_0) = [\![Z]\!](\sigma_1, i_1), \qquad [\![E]\!](\sigma_0, i_0) = [\![E]\!](\sigma_1, i_1), \qquad [\![I]\!](\sigma_0, i_0) = [\![I]\!](\sigma_1, i_1).$$

Proof. The proof is by structural induction.

- **Cases of** $Z \equiv n$ **and** $E \equiv r$**:**
  The desired equalities in both cases follow immediately from the semantics of the target language.
- **Cases of** $Z \equiv x^{\mathsf{int}^\dagger}$ **and** $E \equiv x^{\mathsf{real}^\dagger}$**:**
  The desired conclusions in these cases follow from the assumption that $\sigma_0\ [\mathit{State}_{tgt}(R)]\ \sigma_1$ and $i_0\ [R]\ i_1$.
- **Case of** $Z \equiv \mathrm{op}_i(Z_1, \ldots, Z_n, E_1, \ldots, E_m)$**:**
  By the induction hypothesis, $[\![Z_k]\!](\sigma_0, i_0) = [\![Z_k]\!](\sigma_1, i_1)$ and $[\![E_\ell]\!](\sigma_0, i_0) = [\![E_\ell]\!](\sigma_1, i_1)$ for all $k \in \{1, \ldots, n\}$ and $\ell \in \{1, \ldots, m\}$. Thus, we can prove the desired equality as follows:

  $$\begin{aligned}
  [\![\mathrm{op}_i(Z_1, \ldots, Z_n, E_1, \ldots, E_m)]\!](\sigma_0, i_0) &= [\![\mathrm{op}_i]\!]_a([\![Z_1]\!](\sigma_0, i_0), \ldots, [\![Z_n]\!](\sigma_0, i_0), \\
  &\qquad [\![E_1]\!](\sigma_0, i_0), \ldots, [\![E_m]\!](\sigma_0, i_0)) \\
  &= [\![\mathrm{op}_i]\!]_a([\![Z_1]\!](\sigma_1, i_1), \ldots, [\![Z_n]\!](\sigma_1, i_1), \\
  &\qquad [\![E_1]\!](\sigma_1, i_1), \ldots, [\![E_m]\!](\sigma_1, i_1)) \\
  &= [\![\mathrm{op}_i(Z_1, \ldots, Z_n, E_1, \ldots, E_m)]\!](\sigma_1, i_1).
  \end{aligned}$$

- **Case of** $E \equiv \mathrm{op}_r(Z_1, \ldots, Z_n, E_1, \ldots, E_m)$**:**
  The case can be proved by a similar argument to the one for $Z \equiv \mathrm{op}_i(Z_1, \ldots, Z_n, E_1, \ldots, E_m)$.
- **Case of** $I \equiv [(\alpha_1, Z_1); \ldots; (\alpha_n, Z_n)]$**:**
  By the induction hypothesis, $[\![Z_k]\!](\sigma_0, i_0) = [\![Z_k]\!](\sigma_1, i_1)$ for all $k \in \{1, \ldots, n\}$. Thus,

  $$\begin{aligned}
  [\![[(\alpha_1, Z_1); \ldots; (\alpha_n, Z_n)]]\!](\sigma_0, i_0) &= [(\alpha_1, [\![Z_1]\!](\sigma_0, i_0)); \ldots; (\alpha_n, [\![Z_n]\!](\sigma_0, i_0))] \\
  &= [(\alpha_1, [\![Z_1]\!](\sigma_1, i_1)); \ldots; (\alpha_n, [\![Z_n]\!](\sigma_1, i_1))] \\
  &= [\![[(\alpha_1, Z_1); \ldots; (\alpha_n, Z_n)]]\!](\sigma_1, i_1),
  \end{aligned}$$

  as desired.

□

Let C be a command in the target language that does not use any construct absent in the source language. We define the translation of C to a command $\overline{C}$ in the target language as follows:

$$\begin{aligned}
\overline{x^{\text{real}\dagger} := \textbf{fetch}(I)} &\equiv x^{\text{real}\dagger} := \textbf{fetch}(I), \\
\overline{\textbf{score}(E)} &\equiv \textbf{score}(E), \\
\overline{x^{\text{int}\dagger} := Z} &\equiv x^{\text{int}\dagger} := Z, \\
\overline{x^{\text{real}\dagger} := E} &\equiv x^{\text{real}\dagger} := E, \\
\overline{\textbf{skip}} &\equiv \textbf{skip}, \\
\overline{\textbf{ifz}\ Z\ C_1\ C_2} &\equiv \textbf{ifz}\ Z\ \overline{C_1}\ \overline{C_2}, \\
\overline{C_1; C_2} &\equiv \overline{C_1}; \overline{C_2},
\end{aligned}$$

$$\begin{aligned}
&\overline{\textbf{for}\ x^{\text{int}\dagger}\ \textbf{in}\ \textbf{range}(n)\ \textbf{do}\ C_1} \\
&\quad\equiv\ \textbf{extend_index}(\alpha, n)\ \{ \\
&\qquad \textbf{loop_fixpt_noacc}(n)\ \{ \\
&\qquad\quad \textbf{shift}(\alpha); \\
&\qquad\quad x^{\text{int}\dagger} := \textbf{lookup_index}(\alpha); \\
&\qquad\quad \overline{C_1}\ \}\}
\end{aligned}$$

where $\alpha \in \mathbb{S} \setminus \mathcal{S}(\overline{C_1})$.

Here $\mathcal{S}(C)$ denotes the set of all strings that appear in C.

The following theorem states the soundness of the translation of commands in the target language when the commands are evaluated under the evaluation rules defined by $\Downarrow_{int}$.

Theorem 3. *Let C be a command in the target language that does not use any construct absent in the source language. Let $D \in RDB$, $A_0, A_1 \in FAChain$, $\sigma_0, \sigma_1, \sigma_0', \sigma_1' \in State_{tgt}$, $T_0, T_1 \in Tensor_{\mathbb{R}}$, σ_0 be an A_0-state, and σ_1 be an A_1-state. Suppose that*

$$C, D, \sigma_0, A_0 \Downarrow_{int} \sigma_0', T_0 \quad \textit{and} \quad \overline{C}, D, \sigma_1, A_1 \Downarrow_{int} \sigma_1', T_1$$

Then, for a coupling R such that

$$\sigma_0\ [State_{tgt}(R)]\ \sigma_1 \quad \textit{and} \quad A_0\ [\mathcal{P}(R)]\ A_1,$$

we have that

$$\sigma_0'\ [State_{tgt}(R)]\ \sigma_1' \quad \textit{and} \quad T_0\ [Tensor_{\mathbb{R}}(R)]\ T_1.$$

Proof. We do the case analysis on C.

- **Case of $C \equiv (x^{\text{real}\dagger} := \textbf{fetch}(I))$:**
  By the semantics of commands, $T_0 = T^z_{A_0}$ and $T_1 = T^z_{A_1}$. Since $A_0\ [\mathcal{P}(R)]\ A_1$, by Corollary 1, we have
  $$T_0 = T^z_{A_0}\ [Tensor_{\mathbb{R}}(R)]\ T^z_{A_1} = T_1.$$
  It remains to show that
  $$\sigma_0'\ [State_{tgt}(R)]\ \sigma_1'.$$
  By the semantics of the target language,
  $$\begin{aligned}
  \sigma_0' &= \sigma_0[x^{\text{real}\dagger} : T_0'], & T_0' &= [i_0 \mapsto D([\![I]\!](\sigma_0, i_0)) : i_0 \in A_0], \\
  \sigma_1' &= \sigma_1[x^{\text{real}\dagger} : T_1'], & T_1' &= [i_1 \mapsto D([\![I]\!](\sigma_1, i_1)) : i_1 \in A_1].
  \end{aligned}$$
  We claim that $T_0'\ [Tensor^{\dagger}_{\mathbb{R}}(R)]\ T_1'$. Note that if true, the claim gives the desired conclusion by Lemma 5. Also, note that by Corollary 2, the claim follows if we show the following two conditions:
  (i) $\mathsf{dom}(T_0')\ [\mathcal{P}(R)]\ \mathsf{dom}(T_1')$,
  (ii) $T_0'\ [Tensor_{\mathbb{R}}(R)]\ T_1'$.

The first condition follows immediately from the facts that $\mathsf{dom}(T_0') = A_0$, $\mathsf{dom}(T_1') = A_1$. To prove the second condition, consider arbitrary indices $i_0' \in \mathsf{dom}(T_0')$ and $i_1' \in \mathsf{dom}(T_1')$ with $i_0' \,[R]\, i_1'$. Then, $[\![I]\!](\sigma_0, i_0') = [\![I]\!](\sigma_1, i_1')$ by Proposition 3. Thus,

$$T_0'(i_0') = D([\![I]\!](\sigma_0, i_0')) = D([\![I]\!](\sigma_1, i_1')) = T_1'(i_1'),$$

as desired.

- **Case of** $C \equiv (\mathbf{score}(E))$**:**
  By the semantics of the target language,

$$\sigma_0' = \sigma_0 \qquad T_0 = [i_0 \mapsto [\![E]\!](\sigma_0, i_0) : i_0 \in A_0],$$
$$\sigma_1' = \sigma_1 \qquad T_1 = [i_1 \mapsto [\![E]\!](\sigma_1, i_1) : i_1 \in A_1].$$

  Thus, $\sigma_0' \,[\mathit{State}_{tgt}(R)]\, \sigma_1'$. We claim that $T_0 \,[\mathit{Tensor}_{\mathbb{R}}(R)]\, T_1$. Suppose $i_0 \,[R]\, i_1$. By the assumption $A_0 \,[\mathcal{P}(R)]\, A_1$, we have $\mathsf{dom}(T_0) \,[\mathcal{P}(R)]\, \mathsf{dom}(T_1)$. By Lemma 2, it implies

$$i_0 \in \mathsf{dom}(T_0) \iff i_1 \in \mathsf{dom}(T_1).$$

  Also, by the assumption $\sigma_0 \,[\mathit{State}_{tgt}(R)]\, \sigma_1$ and Proposition 3,

$$T_0(i_0) = [\![E]\!](\sigma_0, i_0) = [\![E]\!](\sigma_1, i_1) = T_1(i_1).$$

  The desired relationship $T_0 \,[\mathit{Tensor}_{\mathbb{R}}(R)]\, T_1$ follows from these two facts.
- **Case of** $C \equiv (x^{\mathsf{int}^\dagger} := Z)$**:**
  By the semantics of the target language,

$$\sigma_0' = \sigma_0[x^{\mathsf{int}^\dagger} : T_0'], \qquad T_0' = [i_0 \mapsto [\![Z]\!](\sigma_0, i_0) : i_0 \in A_0],$$
$$\sigma_1' = \sigma_1[x^{\mathsf{int}^\dagger} : T_1'], \qquad T_1' = [i_1 \mapsto [\![Z]\!](\sigma_1, i_1) : i_1 \in A_1],$$

  and $T_0 = T^z_{A_0}$ and $T_1 = T^z_{A_1}$. Since $A_0 \,[\mathcal{P}(R)]\, A_1$, by Corollary 1, we have

$$T_0 = T^z_{A_0} \,[\mathit{Tensor}_{\mathbb{R}}(R)]\, T^z_{A_1} = T_1.$$

  It remains to prove that $\sigma_0' \,[\mathit{State}_{tgt}(R)]\, \sigma_1'$. Note that by Lemma 5, we can discharge this proof obligation by showing that $T_0' [\mathit{Tensor}^{\dagger}_{\mathbb{R}}(R)] T_1'$. But by Corollary 2, this relationship between T_0' and T_1' follows from the following two conditions:

  (i) $\mathsf{dom}(T_0') \,[\mathcal{P}(R)]\, \mathsf{dom}(T_1')$,
  (ii) $T_0' \,[\mathit{Tensor}_{\mathbb{R}}(R)]\, T_1'$.

  The first condition holds because $\mathsf{dom}(T_0') = A_0$, $\mathsf{dom}(T_1') = A_1$, and $A_0 \,[\mathcal{P}(R)]\, A_1$. To prove the second condition, pick arbitrary $i_0' \in \mathsf{dom}(T_0')$ and $i_1' \in \mathsf{dom}(T_1')$ with $i_0' \,[R]\, i_1'$. Then, since $i_0' \,[R]\, i_1'$ and $\sigma_0 \,[\mathit{State}_{tgt}(R)]\, \sigma_1$, we have $[\![Z]\!](\sigma_0, i_0') = [\![Z]\!](\sigma_1, i_1')$ by Proposition 3. Thus,

$$T_0'(i_0') = [\![Z]\!](\sigma_0, i_0') = [\![Z]\!](\sigma_1, i_1') = T_1'(i_1'),$$

  as desired.
- **Case of** $C \equiv (x^{\mathsf{real}^\dagger} := E)$**:**
  The proof is similar to the one for $C \equiv (x^{\mathsf{int}^\dagger} := Z)$.
- **Case of** $C \equiv (\mathbf{skip})$**:**
  In this case, we have $\sigma_0' = \sigma_0$, $\sigma_1' = \sigma_1$, $T_0 = T^z_{A_0}$, and $T_1 = T^z_{A_1}$. Since $A_0 \,[\mathcal{P}(R)]\, A_1$ and $\sigma_0 \,[\mathit{State}_{tgt}(R)]\, \sigma_1$, by Corollary 1, we have

$$\sigma_0' \,[\mathit{State}_{tgt}(R)]\, \sigma_1' \quad \text{and} \quad T_0 \,[\mathit{Tensor}_{\mathbb{R}}(R)]\, T_1.$$

- **Case of** $C \equiv (C_1; C_2)$**:**
  By the semantics of sequencing, there exist $\sigma_0'', \sigma_1'' \in \mathit{State}_{tgt}$ and $T_0', T_0'', T_1', T_1'' \in \mathit{Tensor}_{\mathbb{R}}$ such that

  $$C_1, D, \sigma_0, A_0 \Downarrow_{int} \sigma_0'', T_0'', \qquad C_2, D, \sigma_0'', A_0 \Downarrow_{int} \sigma_0', T_0', \qquad T_0 = T_0'' \oplus T_0',$$
  $$\overline{C_1}, D, \sigma_1, A_1 \Downarrow_{int} \sigma_1'', T_1'', \qquad \overline{C_2}, D, \sigma_1'', A_1 \Downarrow_{int} \sigma_1', T_1', \qquad T_1 = T_1'' \oplus T_1'.$$

  By the induction hypothesis on C_1, we have

  $$\sigma_0'' \, [\mathit{State}_{tgt}(R)] \, \sigma_1'' \quad \text{and} \quad T_0'' \, [\mathit{Tensor}_{\mathbb{R}}(R)] \, T_1''.$$

  The first relationship from above enables us to apply the induction hypothesis to C_2, and to obtain

  $$\sigma_0' \, [\mathit{State}_{tgt}(R)] \, \sigma_1' \quad \text{and} \quad T_0' \, [\mathit{Tensor}_{\mathbb{R}}(R)] \, T_1'.$$

  Thus, by Lemma 4,

  $$\sigma_0' \, [\mathit{State}_{tgt}(R)] \, \sigma_1' \quad \text{and} \quad T_0 = \left(T_0'' \oplus T_0'\right) [\mathit{Tensor}_{\mathbb{R}}(R)] \left(T_1'' \oplus T_1'\right) = T_1.$$

- **Case of** $C \equiv (\textbf{ifz}\ Z\ C_1\ C_2)$**:**
  Let

  $$A_0' = \{i_0 \in A_0 : [\![Z]\!](\sigma_0, i_0) = 0\}, \qquad A_0'' = \{i_0 \in A_0 : [\![Z]\!](\sigma_0, i_0) \neq 0\},$$
  $$A_1' = \{i_1 \in A_1 : [\![Z]\!](\sigma_1, i_1) = 0\}, \qquad A_1'' = \{i_1 \in A_1 : [\![Z]\!](\sigma_1, i_1) \neq 0\}.$$

  By the semantics, there exist $\sigma_0'', \sigma_1'' \in \mathit{State}_{tgt}$ and $T_0', T_0'', T_1', T_1'' \in \mathit{Tensor}_{\mathbb{R}}$ such that

  $$C_1, D, \sigma_0, A_0' \Downarrow_{int} \sigma_0'', T_0'', \qquad C_2, D, \sigma_0'', A_0'' \Downarrow_{int} \sigma_0', T_0', \qquad T_0 = T_0'' \oplus T_0',$$
  $$\overline{C_1}, D, \sigma_1, A_1' \Downarrow_{int} \sigma_1'', T_1'', \qquad \overline{C_2}, D, \sigma_1'', A_1'' \Downarrow_{int} \sigma_1', T_1', \qquad T_1 = T_1'' \oplus T_1'.$$

  By Lemma 6,

  $$\mathsf{dom}(T_0'') = A_0', \quad \mathsf{dom}(T_0') = A_0'', \quad \mathsf{dom}(T_1'') = A_1', \quad \mathsf{dom}(T_1') = A_1''.$$

  We claim that

  $$A_0' \, [\mathcal{P}(R)] \, A_1' \quad \text{and} \quad A_0'' \, [\mathcal{P}(R)] \, A_1''.$$

  We only prove the first relationship in the claim; the second can be proved similarly. Consider arbitrary indices $i_0, i_1 \in \mathit{Index}$ with $i_0 \, [R] \, i_1$. We need to prove that

  (i) $i_0 \in A_0'{\uparrow} \iff i_1 \in A_1'{\uparrow}$,
  (ii) $i_0 \in A_0'{\uparrow} \implies \max(A_0' \cap i_0{\downarrow}) \, [R] \, \max(A_1' \cap i_1{\downarrow})$.

  First, we show the only-if direction of the first condition; the if direction can be proved similarly. Assume that $i_0 \in A_0'{\uparrow}$. Then, $i_0 \in A_0{\uparrow}$ and $i_1 \in A_1{\uparrow}$ since $A_0' \subseteq A_0$ and $A_0 \, [\mathcal{P}(R)] \, A_1$. Thus, since $A_1{\uparrow} = A_1'{\uparrow} \cup A_1''{\uparrow}$, it suffices to show that $i_1 \notin A_1''{\uparrow}$. For the sake of contradiction, suppose that $i_1 \in A_1''{\uparrow}$. Then, again since $A_0 \, [\mathcal{P}(R)] \, A_1$,

  $$\max(A_0' \cap i_0{\downarrow}) = \max(A_0 \cap i_0{\downarrow}) \, [R] \, \max(A_1 \cap i_1{\downarrow}) = \max(A_1'' \cap i_1{\downarrow}).$$

  Since $\sigma_0 \, [\mathit{State}_{tgt}(R)] \, \sigma_1$, by Proposition 3, we have

  $$[\![Z]\!](\sigma_0, \max(A_0' \cap i_0{\downarrow})) = [\![Z]\!](\sigma_1, \max(A_1'' \cap i_1{\downarrow})).$$

  But this cannot happen because the left hand side of the equation is 0 and the right hand side is not 0. Next, we prove the second condition. Assume that $i_0 \in A_0'{\uparrow}$, and so $i_1 \in A_1'{\uparrow}$. Then, since $A_0 \, [\mathcal{P}(R)] \, A_1$ and $A_0' \subseteq A_0$ and $A_1' \subseteq A_1$ and both A_0 and A_1 are antichains, we have

  $$\max(A_0' \cap i_0{\downarrow}) = \max(A_0 \cap i_0{\downarrow}) \, [R] \, \max(A_1 \cap i_1{\downarrow}) = \max(A_1' \cap i_1{\downarrow}),$$

and thus, the claim holds. By $A_0' \, [\mathcal{P}(R)] \, A_1'$ and the assumption $\sigma_0 \, [State_{tgt}(R)] \, \sigma_1$, we can apply the induction hypothesis to C_1. This gives us

$$\sigma_0'' \, [State_{tgt}(R)] \, \sigma_1'' \quad \text{and} \quad T_0'' \, [Tensor_{\mathbb{R}}(R)] \, T_1''.$$

The first conjunct from above and $A_0'' \, [\mathcal{P}(R)] \, A_1''$ let us apply the induction hypothesis to C_2, which gives us

$$\sigma_0' \, [State_{tgt}(R)] \, \sigma_1' \quad \text{and} \quad T_0' \, [Tensor_{\mathbb{R}}(R)] \, T_1'.$$

By Lemma 4, we have

$$T_0 = (T_0'' \oplus T_0') \, [Tensor_{\mathbb{R}}(R)] \, (T_1'' \oplus T_1') = T_1.$$

- **Case of** $C \equiv (\textbf{for } x^{\mathsf{int}^\dagger} \textbf{ in range}(n) \textbf{ do } C')$**:**

  Dynamics of C**.** By the semantics of the sequential composition and the assignment command, there exist

  $$\sigma_{(0,k)}, \sigma'_{(0,k)} \in State_{tgt} \quad \text{and} \quad T_{(0,k)} \in Tensor_{\mathbb{R}} \quad \text{for all } k \in \{0, \ldots, n-1\}$$

  such that for all $k \in \{0, \ldots, n-1\}$,

  $$\sigma'_{(0,k)} = \sigma_{(0,k-1)}[x^{\mathsf{int}^\dagger} : [i \mapsto k : i \in A_0]], \qquad C', D, \sigma'_{(0,k)}, A_0 \Downarrow_{int} \sigma_{(0,k)}, T_{(0,k)},$$

  where

  $$\sigma_{(0,-1)} = \sigma_0, \qquad \sigma_{(0,n-1)} = \sigma_0', \qquad T_0 = \left[i \mapsto \sum_{k=0}^{n-1} T_{(0,k)}(i) \;:\; i \in A_0 \right].$$

  Dynamics of $\overline{C}$**.** Let α be a fresh string such that $\alpha \notin \mathrm{dom}(i)$ for all $i \in A_0 \cup A_1$. We will use α to translate C to $\overline{C}$. Define

  $$A_1' = \{i +\!\!+ [(\alpha, k)] : i \in A_1 \text{ and } k \in \{0, \ldots, n-1\}\}.$$

  Let us use _ to denote the real tensor $T^z_{A_1'}$. Then, by the semantics of the target language, there exist

  $$\sigma_{(1,k)}, \sigma'_{(1,k)}, \sigma''_{(1,k)} \in State_{tgt} \quad \text{and} \quad T_{(1,k)} \in Tensor_{\mathbb{R}} \quad \text{for all } k \in \{0, \ldots, n-1\}$$

  such that

  (1) for all $k \in \{0, \ldots, n-1\}$,

  $$\begin{aligned} \textbf{shift}(\alpha), D, \sigma_{(1,k-1)}, A_1' &\Downarrow_{int} \sigma'_{(1,k)}, \_, \\ (x^{\mathsf{int}^\dagger} := \textbf{lookup\_index}(\alpha)), D, \sigma'_{(1,k)}, A_1' &\Downarrow_{int} \sigma''_{(1,k)}, \_, \\ \overline{C'}, D, \sigma''_{(1,k)}, A_1' &\Downarrow_{int} \sigma_{(1,k)}, T_{(1,k)}, \end{aligned}$$

  (2) for all $k \in \{0, \ldots, n-1\}$,

  $$\rho_1'(i) = \begin{cases} i +\!\!+ [(\alpha, 0)] & \text{if } \alpha \notin \mathrm{dom}(i) \text{ and } i +\!\!+ [(\alpha, 0)] \in A_1', \\ j +\!\!+ [(\alpha, m+1)] & \text{if } i = j +\!\!+ [(\alpha, m)] \in A_1' \downarrow \\ & \text{and } j +\!\!+ [(\alpha, m+1)] \in A_1' \text{ for some } j \in Index \text{ and } m \geq 0, \\ \text{undefined} & \text{otherwise,} \end{cases}$$

  $$\sigma_{(1,-1)} = \sigma_1, \qquad \sigma'_{(1,k)} = \sigma_{(1,k-1)}\langle \rho_1' \rangle, \qquad \sigma''_{(1,k)} = \sigma'_{(1,k)}[x^{\mathsf{int}^\dagger} : [i \mapsto i(\alpha) : i \in A_1']].$$

(3) finally,

$$\rho_1 = [i +\!\!+ [(\alpha, n-1)] \mapsto i : i \in A_1], \qquad T_1 = \left[i \mapsto \sum_{k=0}^{n-1} T_{(1,n-1)}(i +\!\!+ [(\alpha,k)]) : i \in A_1 \right].$$
$$\sigma_1' = \sigma_{(1,n-1)}\langle \rho_1 \rangle,$$

Overview. The rest of the proof is about relating the executions of C and $\overline{C}$ and deriving the conclusion of the theorem. For this purpose, we define relations $R^{(-1)}$ and $R^{(k)}$ on indices for all $k \in \{0, \ldots, n-1\}$ as follows: for all $i_0, i_1, i_1' \in \mathit{Index}$,

$$i_0 \, [R^{(-1)}] \, i_1 \iff i_0 \in A_0 \text{ and } i_1 \in A_1 \text{ and } i_0 \, [R] \, i_1,$$
$$i_0 \, [R^{(k)}] \, i_1' \iff \exists \, i_1 \in A_1. \left(i_0 \, [R^{(-1)}] \, i_1 \text{ and } i_1' = i_1 +\!\!+ [(\alpha, k)] \right).$$

Then, $R^{(-1)}$ and $R^{(k)}$ are couplings for all $k \in \{0, \ldots, n-1\}$. Our proof relies on the application of the induction hypothesis to C' and $\overline{C'}$. Our proof consists of three parts.

(1) The first part shows that

$$A_0 \, [\mathcal{P}(R^{(k)})] \, A_1' \quad \text{for all } k \in \{0, 1, \ldots, n-1\}.$$

(2) The second part shows that for all $k \in \{0, \ldots, n-1\}$ and $\ell \in \{0, \ldots, k\}$,

$$\sigma_0 \, [\mathit{State}_{tgt}(R^{(-1)})] \, \sigma''_{(1,k)}, \quad \text{and} \quad \sigma'_{(0,\ell)} \, [\mathit{State}_{tgt}(R^{(\ell)})] \, \sigma''_{(1,k)}.$$

(3) The third part derives the claimed conclusion of the theorem.

First part. We spell out the first part of the proof, namely,

$$A_0 \, [\mathcal{P}(R^{(k)})] \, A_1' \quad \text{for all } k \in \{0, \ldots, n-1\}.$$

Pick $k \in \{0, \ldots, n-1\}$, indices i_0, i_1' such that $i_0 \, [R^{(k)}] \, i_1'$ arbitrarily. Then,

$$\exists \, i_1 \in A_1. \left(i_0 \, [R^{(-1)}] \, i_1 \text{ and } i_1' = i_1 +\!\!+ [(\alpha, k)] \right).$$

We only need to show that

$$\max(A_0 \cap i_0{\downarrow}) \, [R^{(k)}] \, \max(A_1' \cap i_1'{\downarrow}).$$

This is obvious since $\max(A_0 \cap i_0{\downarrow}) = i_0$ and $\max(A_1' \cap i_1'{\downarrow}) = i_1'$.

Second part. We describe the second part of our proof:

$$\sigma_0 \, [\mathit{State}_{tgt}(R^{(-1)})] \, \sigma''_{(1,k)}, \quad \text{and} \quad \sigma'_{(0,\ell)} \, [\mathit{State}_{tgt}(R^{(\ell)})] \, \sigma''_{(1,k)} \quad \text{for all } \ell \in \{0, \ldots, k\}.$$

The proof is based on the inductive argument on k.

Base case. The base case is that

$$\sigma_0 \, [\mathit{State}_{tgt}(R^{(-1)})] \, \sigma''_{(1,0)} \quad \text{and} \quad \sigma'_{(0,0)} \, [\mathit{State}_{tgt}(R^{(0)})] \, \sigma''_{(1,0)}.$$

Recall the definitions of $\sigma''_{(1,0)}$ and $\sigma'_{(0,0)}$:

$$\begin{aligned} \sigma''_{(1,0)} &= \sigma'_{(1,0)}[x^{\mathsf{int}^\dagger} : [i \mapsto i(\alpha) : i \in A_1']] \\ &= \sigma_1\langle \rho_1' \rangle [x^{\mathsf{int}^\dagger} : [i \mapsto i(\alpha) : i \in A_1']], \\ \sigma'_{(0,0)} &= \sigma_0[x^{\mathsf{int}^\dagger} : [i \mapsto 0 : i \in A_0]]. \end{aligned}$$

To show the first relationship from above, pick a variable $y^{\tau^\dagger}$ and indices i_0 and i_1 with $i_0 \, [R^{(-1)}] \, i_1$ arbitrarily. Then, $i_0 \in A_0$ and $i_1 \in A_1$ and $i_0 \, [R] \, i_1$. These imply that

$$\mathit{extend}(\sigma''_{(1,0)}(y^{\tau^\dagger}))(i_1) = \mathit{extend}(\sigma_1\langle \rho_1' \rangle [x^{\mathsf{int}^\dagger} : [i \mapsto i(\alpha) : i \in A_1']](y^{\tau^\dagger}))(i_1)$$

$$= extend(\sigma_1\langle\rho_1'\rangle(y^{\tau\dagger}))(i_1)$$
$$= extend(\sigma_1(y^{\tau\dagger}))(i_1)$$
$$= extend(\sigma_0(y^{\tau\dagger}))(i_0).$$

The second equality holds by Lemma 1 because $i_1 \in A_1$ implies $i_1 \notin A_1'\uparrow$. The third equality also holds by Lemma 1 because $\mathsf{image}(\rho_1')\uparrow \subseteq A_1'\uparrow$ and $i_1 \notin A_1'\uparrow$ imply $i_1 \notin \mathsf{image}(\rho_1')\uparrow$. To prove the second relationship from above, pick a variable $y^{\tau\dagger}$ and indices i_0, i_1' with $i_0\ [R^{(0)}]\ i_1'$ arbitrarily. Then, $i_0 \in A_0$ and $i_1' \in A_1'$ and

$$\exists\, i_1 \in A_1.\ \big(i_1' = i_0\ [R]\ i_1 \text{ and } i_1' = i_1 +\!\!+ [(\alpha, 0)]\big).$$

We have to show that

$$extend(\sigma'_{(0,0)}(y^{\tau\dagger}))(i_0) = extend(\sigma''_{(1,0)}(y^{\tau\dagger}))(i_1).$$

We discharge this proof obligation as follows:

$$\begin{aligned}
extend(\sigma''_{(1,0)}(y^{\tau\dagger}))(i_1') &= extend(\sigma_1\langle\rho_1'\rangle[x^{\mathsf{int}\dagger} : [i \mapsto i(\alpha) : i \in A_1']](y^{\tau\dagger}))(i_1') \\
&= \begin{cases} extend([i \mapsto i(\alpha) : i \in A_1'])(i_1') & \text{if } y^{\tau\dagger} \equiv x^{\mathsf{int}\dagger} \\ extend(\sigma_1\langle\rho_1'\rangle(y^{\tau\dagger}))(i_1') & \text{otherwise} \end{cases} \\
&= \begin{cases} 0 & \text{if } y^{\tau\dagger} \equiv x^{\mathsf{int}\dagger} \\ extend(\sigma_1(y^{\tau\dagger}))(i_1) & \text{otherwise} \end{cases} \\
&= \begin{cases} extend([i \mapsto 0 : i \in A_0])(i_0) & \text{if } y^{\tau\dagger} \equiv x^{\mathsf{int}\dagger} \\ extend(\sigma_0(y^{\tau\dagger}))(i_0) & \text{otherwise} \end{cases} \\
&= extend(\sigma[x^{\mathsf{int}\dagger} : [i \mapsto 0 : i \in A_0]](y^{\tau\dagger}))(i_0) \\
&= extend(\sigma'_{(0,0)}(y^{\tau\dagger}))(i_0).
\end{aligned}$$

The second equality holds by Lemma 1 because $i_1' \in A_1'$. The third equality also holds by Lemma 1 because $\rho_1'(i_1) = i_1 +\!\!+ [(\alpha, 0)] = i_1'$. The fourth equality holds because $i_0\ [R]\ i_1$ and $\sigma_0\ [State_{tgt}(R)]\ \sigma_1$. The fifth equality holds by Lemma 1 because $i_0 \in A_0$.

Inductive case. Let's move on to the inductive case. Consider $k \in \{1, \ldots, n-1\}$. Assume that

$$\sigma_0\ [State_{tgt}(R^{(-1)})]\ \sigma''_{(1,k-1)}, \quad \text{and} \quad \sigma'_{(0,\ell)}\ [State_{tgt}(R^{(\ell)})]\ \sigma''_{(1,k-1)} \quad \text{for all } \ell \in \{0, \ldots, k-1\}.$$

We have to show that

$$\sigma_0\ [State_{tgt}(R^{(-1)})]\ \sigma''_{(1,k)}, \quad \text{and} \quad \sigma'_{(0,\ell)}\ [State_{tgt}(R^{(\ell)})]\ \sigma''_{(1,k)} \quad \text{for all } \ell \in \{0, \ldots, k\}.$$

Recall the definitions of $\sigma''_{(1,k)}$ and $\sigma'_{(0,\ell)}$:

$$\begin{aligned}
\sigma''_{(1,k)} &= \sigma'_{(1,k)}[x^{\mathsf{int}\dagger} : [i \mapsto i(\alpha) : i \in A_1']] \\
&= \sigma_{(1,k-1)}\langle\rho_1'\rangle[x^{\mathsf{int}\dagger} : [i \mapsto i(\alpha) : i \in A_1']], \\
\sigma'_{(0,\ell)} &= \sigma_{(0,\ell-1)}[x^{\mathsf{int}\dagger} : [i \mapsto \ell : i \in A_0]].
\end{aligned}$$

To prove $\sigma_0\ [State_{tgt}(R^{(-1)})]\ \sigma''_{(1,k)}$, pick a variable $y^{\tau\dagger}$ and indices i_0, i_1 with $i_0\ [R^{(-1)}]\ i_1$. Then, $i_0 \in A_0$ and $i_1 \in A_1$ and $i_0\ [R]\ i_1$. These imply that

$$\begin{aligned}
extend(\sigma''_{(1,k)}(y^{\tau\dagger}))(i_1) &= extend(\sigma_{(1,k-1)}\langle\rho_1'\rangle[x^{\mathsf{int}\dagger} : [i \mapsto i(\alpha) : i \in A_1']](y^{\tau\dagger}))(i_1) \\
&= extend(\sigma_{(1,k-1)}\langle\rho_1'\rangle(y^{\tau\dagger}))(i_1)
\end{aligned}$$

$$= extend(\sigma_{(1,k-1)}(y^{\tau^\dagger}))(i_1)$$
$$= extend(\sigma''_{(1,k-1)}(y^{\tau^\dagger}))(i_1)$$
$$= extend(\sigma_0(y^{\tau^\dagger}))(i_0).$$

The second equality holds by Lemma 1 because $i_1 \in A_1$ implies $i_1 \notin A'_1\uparrow$. The third equality also holds by Lemma 1 because $\mathrm{image}(\rho'_1)\uparrow \subseteq A'_1\uparrow$ and $i_1 \notin A'_1\uparrow$ imply $i_1 \notin \mathrm{image}(\rho'_1)\uparrow$. The fourth equality holds because $\sigma''_{(1,k-1)} =_{(A'_1\uparrow)^c} \sigma_{(1,k-1)}$ by Lemma 6 and $i_1 \notin A'_1\uparrow$. The fifth equality holds because

$$i_0\ [R^{(-1)}]\ i_1 \quad \text{and} \quad \sigma_0\ [State_{tgt}(R^{(-1)})]\ \sigma''_{(1,k-1)}.$$

To prove that $\sigma'_{(0,\ell)}\ [State_{tgt}(R^{(\ell)})]\ \sigma''_{(1,k)}$, pick a variable $y^{\tau^\dagger}$ and indices i_0, i'_1 with $i_0\ [R^{(\ell)}]\ i'_1$ arbitrarily. Then, $i_0 \in A_0$ and $i'_1 \in A'_1$ and

$$\exists\, i_1 \in A_1.\ \big(i'_1 = i_0\ [R]\ i_1 \text{ and } i'_1 = i_1 +\!\!+ [(\alpha, \ell)]\big).$$

If $\ell = 0$, then

$$\begin{aligned}
extend(\sigma''_{(1,k)}(y^{\tau^\dagger}))(i'_1) &= extend(\sigma_{(1,k-1)}\langle\rho'_1\rangle[x^{\mathsf{int}^\dagger} : [i \mapsto i(\alpha) : i \in A'_1]](y^{\tau^\dagger}))(i'_1) \\
&= \begin{cases} extend([i \mapsto i(\alpha) : i \in A'_1])(i'_1) & \text{if } y^{\tau^\dagger} \equiv x^{\mathsf{int}^\dagger} \\ extend(\sigma_{(1,k-1)}\langle\rho'_1\rangle(y^{\tau^\dagger}))(i'_1) & \text{otherwise} \end{cases} \\
&= \begin{cases} 0 & \text{if } y^{\tau^\dagger} \equiv x^{\mathsf{int}^\dagger} \\ extend(\sigma_{(1,k-1)}(y^{\tau^\dagger}))(i_1) & \text{otherwise} \end{cases} \\
&= \begin{cases} 0 & \text{if } y^{\tau^\dagger} \equiv x^{\mathsf{int}^\dagger} \\ extend(\sigma''_{(1,k-1)}(y^{\tau^\dagger}))(i_1) & \text{otherwise} \end{cases} \\
&= \begin{cases} 0 & \text{if } y^{\tau^\dagger} \equiv x^{\mathsf{int}^\dagger} \\ extend(\sigma_0(y^{\tau^\dagger}))(i_0) & \text{otherwise} \end{cases} \\
&= extend(\sigma_0[x^{\mathsf{int}^\dagger} : [i \mapsto 0 : i \in A_0]](y^{\tau^\dagger}))(i_0) \\
&= extend(\sigma'_{(0,0)}(y^{\tau^\dagger}))(i_0).
\end{aligned}$$

The second equality holds by Lemma 1 because $i'_1 \in A'_1\uparrow$. The third equality also holds by Lemma 1 because

$$i'_1 = i_1 +\!\!+ [(\alpha, 0)] = \rho'_1(i_1).$$

The fourth equality holds because $\sigma''_{(1,k-1)} =_{(A'_1\uparrow)^c} \sigma_{(1,k-1)}$ by Lemma 6 and $i_1 \notin A'_1\uparrow$. The fifth equality holds because

$$i_0\ [R^{(-1)}]\ i_1 \quad \text{and} \quad \sigma_0\ [State_{tgt}(R^{(-1)})]\ \sigma''_{(1,k-1)}.$$

The sixth equality holds by Lemma 1 because $i_0 \in A_0\uparrow$. If $\ell \geq 1$, then

$$\begin{aligned}
extend(\sigma''_{(1,k)}(y^{\tau^\dagger}))(i'_1) &= extend(\sigma_{(1,k-1)}\langle\rho'_1\rangle[x^{\mathsf{int}^\dagger} : [i \mapsto i(\alpha) : i \in A'_1]](y^{\tau^\dagger}))(i'_1) \\
&= \begin{cases} extend([i \mapsto i(\alpha) : i \in A'_1])(i'_1) & \text{if } y^{\tau^\dagger} \equiv x^{\mathsf{int}^\dagger} \\ extend(\sigma_{(1,k-1)}\langle\rho'_1\rangle(y^{\tau^\dagger}))(i'_1) & \text{otherwise} \end{cases} \\
&= \begin{cases} \ell & \text{if } y^{\tau^\dagger} \equiv x^{\mathsf{int}^\dagger} \\ extend(\sigma_{(1,k-1)}(y^{\tau^\dagger}))(i_1 +\!\!+ [(\alpha, \ell-1)]) & \text{otherwise} \end{cases}
\end{aligned}$$

$$= \begin{cases} \ell & \text{if } y^{\tau\dagger} \equiv x^{\mathsf{int}\dagger} \\ extend(\sigma_{(0,\ell-1)}(y^{\tau\dagger}))(i_0) & \text{otherwise} \end{cases}$$
$$= extend(\sigma_{(0,\ell-1)}[x^{\mathsf{int}\dagger} : [i \mapsto \ell : i \in A_0]](y^{\tau\dagger}))(i_0)$$
$$= extend(\sigma'_{(0,\ell)}(y^{\tau\dagger}))(i_0).$$

The second equality holds by Lemma 1 because $i'_1 \in A'_1\uparrow$. The third equality also holds by Lemma 1 because

$$i'_1 = i_1 +\!\!+ [(\alpha, \ell)] = \rho'_1(i_1 +\!\!+ [(\alpha, \ell-1)]).$$

The fifth equality also holds by Lemma 1 because $i_0 \in A_0\uparrow$. The only nontrivial part is the fourth equality. The fourth equality follows from that

$$i_0\ [R^{(\ell-1)}]\ (i_1 +\!\!+ [(\alpha, \ell-1)]) \quad \text{and} \quad \sigma_{(0,\ell-1)}\ [State_{tgt}(R^{(\ell-1)})]\ \sigma_{(1,k-1)}.$$

The second relationship holds due to the following reasons:

$$C', D, \sigma'_{(0,\ell-1)}, A_0 \Downarrow_{int} \sigma_{(0,\ell-1)}, T_{(0,\ell-1)}, \qquad \overline{C'}, D, \sigma''_{(1,k-1)}, A'_1 \Downarrow_{int} \sigma_{(1,k-1)}, T_{(1,k-1)},$$

$\sigma'_{(0,\ell-1)}\ [State_{tgt}(R^{(\ell-1)})]\ \sigma''_{(1,k-1)}$, and $A_0\ [\mathcal{P}(R^{(\ell-1)})]\ A'_1$, which are sufficient conditions to apply the induction hypothesis of the theorem on C'.

Third part. Applying $k = n-1$ to the result of the second part, we have

$$\sigma'_{(0,\ell)}\ [State_{tgt}(R^{(\ell)})]\ \sigma''_{(1,n-1)} \quad \text{for all } \ell \in \{0, \ldots, n-1\}.$$

By the definition, we have

$$C', D, \sigma'_{(0,\ell)}, A_0 \Downarrow_{int} \sigma_{(0,\ell)}, T_{(0,\ell)}, \quad \text{and} \quad \overline{C'}, D, \sigma''_{(1,n-1)}, A'_1 \Downarrow_{int} \sigma_{(1,n-1)}, T_{(1,n-1)}.$$

By the induction hypothesis on C' and the fact $A_0\ [\mathcal{P}(R^{(\ell)})]\ A'_1$ from the first part, we have

$$\sigma_{(0,\ell)}\ [State_{tgt}(R^{(\ell)})]\ \sigma_{(1,n-1)} \quad \text{and} \quad T_{(0,\ell)}\ [Tensor_{\mathbb{R}}(R^{(\ell)})]\ T_{(1,n-1)} \quad \text{for all } \ell \in \{0, \ldots, n-1\}.$$

In particular, $\ell = n-1$ gives $\sigma_{(0,n-1)}\ [State_{tgt}(R^{(n-1)})]\ \sigma_{(1,n-1)}$. By using this, we want to show

$$\sigma'_0\ [State_{tgt}(R)]\ \sigma'_1.$$

Recall the definitions of σ'_0 and σ'_1:

$$\sigma'_0 = \sigma_{(0,n-1)} \quad \text{and} \quad \sigma'_1 = \sigma_{(1,n-1)}\langle\rho_1\rangle.$$

Let $y^{\tau\dagger}$ be a variable and i_0, i_1 be indices such that $i_0\ [R]\ i_1$. We first assume $i_0 \in A_0\uparrow$. Then, $A_0\ [\mathcal{P}(R)]\ A_1$ implies $i_1 \in A_1\uparrow$. This also implies that

$$\begin{aligned} extend(\sigma'_0(y^{\tau\dagger}))(i_0) &= extend(\sigma'_0(y^{\tau\dagger}))(\max(A_0 \cap i_0\downarrow)) \\ &= extend(\sigma_{(0,n-1)}(y^{\tau\dagger}))(\max(A_0 \cap i_0\downarrow)) \\ &= extend(\sigma_{(1,n-1)}(y^{\tau\dagger}))(\max(A_1 \cap i_1\downarrow) +\!\!+ [(\alpha, n-1)]) \\ &= extend(\sigma_{(1,n-1)}\langle\rho_1\rangle(y^{\tau\dagger}))(\max(A_1 \cap i_1\downarrow)) \\ &= extend(\sigma'_1(y^{\tau\dagger}))(i_1). \end{aligned}$$

The first and the last equalities hold because σ'_0 is an A_0-state and σ'_1 is an A_1-state by Lemma 9. The third equality holds because

$$\begin{aligned} A_0\ [\mathcal{P}(R)]\ A_1 &\implies \max(A_0 \cap i_0\downarrow)\ [R]\ \max(A_1 \cap i_1\downarrow) \\ &\implies \max(A_0 \cap i_0\downarrow)\ [R^{(-1)}]\ \max(A_1 \cap i_1\downarrow) \end{aligned}$$

$$\Longrightarrow \max(A_0 \cap i_0{\downarrow})\ [R^{(n-1)}]\ \max(A_1 \cap i_1{\downarrow}) +\!\!+ [(\alpha, n-1)].$$

and $\sigma_{(0,n-1)}\ [State_{tgt}(R^{(n-1)})]\ \sigma_{(1,n-1)}$. The fourth equality holds by Lemma 1 because

$$\rho_1(\max(A_1 \cap i_1{\downarrow}) +\!\!+ [(\alpha, n-1)]) = \max(A_1 \cap i_1{\downarrow}).$$

Now, we assume $i_0 \notin A_0{\uparrow}$ and $i_1 \notin A_1{\uparrow}$. Then, by Lemma 6, we have

$$\sigma_0 =_{(A_0{\uparrow})^c} \sigma_0', \quad \text{and} \quad \sigma_1 =_{(A_1{\uparrow})^c} \sigma_1'.$$

This implies that

$$\begin{aligned} extend(\sigma_0'(y^{\tau^\dagger}))(i_0) &= extend(\sigma_0(y^{\tau^\dagger}))(i_0) \\ &= extend(\sigma_1(y^{\tau^\dagger}))(i_1) \\ &= extend(\sigma_1'(y^{\tau^\dagger}))(i_1). \end{aligned}$$

The second equality holds because

$$\sigma_0\ [State_{tgt}(R)]\ \sigma_1 \quad \text{and} \quad i_0\ [R]\ i_1.$$

It remains to show the other conclusion of the theorem about T_0 and T_1:

$$T_0\ [Tensor_{\mathbb{R}}(R)]\ T_1.$$

Note that $\mathrm{dom}(T_0) = A_0$ and $\mathrm{dom}(T_1) = A_1$ by Lemma 6. For all $i_0 \in A_0$ and $i_1 \in A_1$, if $i_0\ [R]\ i_1$, then

$$T_0(i_0) = \sum_{\ell=0}^{n-1} T_{(0,\ell)}(i_0) = \sum_{\ell=0}^{n-1} T_{(1,n-1)}(i_1 +\!\!+ [(\alpha, \ell)]) = T_1(i_1),$$

where the second equality holds since $T_{(0,\ell)}\ [Tensor_{\mathbb{R}}(R^{(\ell)})]\ T_{(1,n-1)}$ and $i_0\ [R^{(\ell)}]\ (i_1 +\!\!+ [(\alpha, \ell)])$ for all $\ell \in \{0, \ldots, n-1\}$.

□

A.5 Soundness of the Parallelising Translation from Source to Target Language

In this subsection, we show the soundness of our parallelising translation. The following lemma states that the command C in the source language and its parallelising translation $\overline{C}$ in the target language always successfully terminate where the latter one is run with an antichain $\{[]\}$.

Lemma 10. *Let C be a command in the source language, and $\overline{C}$ be the parallelising translation of C in the target language. Let $D \in RDB$, $\sigma \in State_{src}$, $\overline{\sigma} \in State_{tgt}$, and $A \in FAChain$ such that*

$$\mathcal{S}(\overline{C}) \cap \mathrm{dom}(i) = \emptyset \text{ for all } i \in A.$$

Then, there exist $\sigma' \in State_{src}$, $\overline{\sigma}' \in State_{tgt}$, $r \in \mathbb{R}$, and $T \in Tensor_{\mathbb{R}}$ such that

$$C, D, \sigma \Downarrow \sigma', r \quad \text{and} \quad \overline{C}, D, \overline{\sigma}, A \Downarrow_{tgt} \overline{\sigma}', T.$$

Proof. The proofs are based on structural induction on C.

(1) the source language
Since every expression and update on an arbitrary state doesn't occur any error, we can always find the proper σ', r for every syntax which has a rule with no premises by calculating the result of the rule. Therefore, what we need is to prove the case for the rules with premises.

(a) $C \equiv (C_1; C_2)$:
By the induction hypothesis applying to C_1 and C_2, there exist σ_1, r_1 such that $C_1, D, \sigma \Downarrow \sigma_1, r_1$, and also there exist σ_2, r_2 such that $C_2, D, \sigma_1 \Downarrow \sigma_2, r_2$. Therefore, $(C_1; C_2), D, \sigma \Downarrow \sigma_2, r_1 + r_2$.

(b) $C \equiv (\textbf{ifz}\ Z\ C_1\ C_2)$:
By the induction hypothesis applying to C_1 and C_2, there exist σ_1, r_1 such that $C_1, D, \sigma \Downarrow \sigma_1, r_1$, and also there exist σ_2, r_2 such that $C_2, D, \sigma \Downarrow \sigma_2, r_2$. If $[\![Z]\!]\sigma = 0$, we can apply the first rule for **ifz**, so that $(\textbf{ifz}\ Z\ C_1\ C_2), D, \sigma \Downarrow \sigma_1, r_1$. Otherwise, $[\![Z]\!]\sigma \neq 0$, then we can apply the second rule for **ifz**, so that $(\textbf{ifz}\ Z\ C_1\ C_2), D, \sigma \Downarrow \sigma_2, r_2$.

(c) $C \equiv (\textbf{for}\ x\ \textbf{in}\ \textbf{range}(n)\ \textbf{do}\ C')$:
By the induction hypothesis applying to C', there exist σ_i, r_i for $1 <= i <= n$ such that $C', D, \sigma_{i-1}[x \mapsto i-1] \Downarrow \sigma_i, r_i$ for all i, where $\sigma_0 = \sigma$.
Therefore, $(\textbf{for}\ x\ \textbf{in}\ \textbf{range}(n)\ \textbf{do}\ C'), D, \sigma_0 \Downarrow \sigma_n, \sum_{k=1}^{n} r_k$

(2) the target language
Similar to the source language, every expression and update on an arbitrary state and an arbitrary tensor are well-defined. Therefore, there exist $\overline{\sigma}', T$ for every syntax which has a rule with assignment premises, which only introduce new symbol for the result of operation or update (e.g. $T = \ldots$).

(a) $C \equiv (C_1; C_2)$:
Since $\mathcal{S}(\overline{C_1}), \mathcal{S}(\overline{C_2}) \subseteq \mathcal{S}(\overline{C})$, we can apply the induction hypothesis to premises. Therefore, there exist $\overline{\sigma_1}$, T_1 such that $\overline{C_1}, D, \overline{\sigma}, A \Downarrow_{tgt} \overline{\sigma_1}, T_1$, and also there exist $\overline{\sigma_2}$, T_2 such that $\overline{C_2}, D, \overline{\sigma_1}, A \Downarrow_{tgt} \overline{\sigma_2}, T_2$. Therefore, $\overline{(C_1; C_2)}, D, \overline{\sigma}, A \Downarrow_{tgt} \overline{\sigma_2}, (T_1 \oplus T_2)$.

(b) $C \equiv (\textbf{ifz}\ Z\ C_1\ C_2)$:
Since $A_1, A_2 \subseteq A$ and $\mathcal{S}(\overline{C_1}), \mathcal{S}(\overline{C_2}) \subseteq \mathcal{S}(\overline{C})$, we can apply the induction hypothesis to premises. Therefore, there exist $\overline{\sigma_1}$, T_1 such that $\overline{C_1}, D, \overline{\sigma}, A_1 \Downarrow_{tgt} \overline{\sigma_1}, T_1$, and also there exist $\overline{\sigma_2}$, T_2 such that $\overline{C_2}, D, \overline{\sigma_1}, A_2 \Downarrow_{tgt} \overline{\sigma_2}, T_2$. Therefore, $\overline{(\textbf{ifz}\ Z\ C_1\ C_2)}, D, \overline{\sigma}, A \Downarrow_{tgt} \overline{\sigma_2}, (T_1 \oplus T_2)$.

(c) $C \equiv (\textbf{for}\ x\ \textbf{in}\ \textbf{range}(n)\ \textbf{do}\ C_1)$:
To simplify the proof, we introduce sub-commands of the translated code as follows:

$$\overline{C_2} \stackrel{\text{def}}{=} x^{\text{int}\dagger} := \textbf{lookup\_index}(\alpha);\ \overline{C_1}$$

$$\overline{C_3} \stackrel{\text{def}}{=} \textbf{shift}(\alpha);\ \overline{C_2}$$

$$\overline{C_4} \stackrel{\text{def}}{=} \textbf{loop\_fixpt\_noacc}(n)\{\ \overline{C_3}\ \}.$$

Then, $\overline{C} \equiv \textbf{extend\_index}(\alpha, n)\{\ \overline{C_4}\ \}$.
We will prove the whole part with top-down approaches. Each part is a proof for a statement which has a form as follows:
For all $\overline{\sigma} \in \mathit{State}_{tgt}$, there exist $\overline{\sigma}', T$ such that $\hat{C}, D, \overline{\sigma}, \hat{A} \Downarrow_{tgt} \overline{\sigma}', T$,
where $\hat{C}, \hat{A}$ are fixed to specific values.

(i) $\hat{C} \equiv (\textbf{extend\_index}(\alpha, n)\{\ \overline{C_4}\ \})$ and $\hat{A} \equiv A$ (the target statement)
Since $\mathcal{S}(\overline{C}) \cap \mathrm{dom}(i) = \emptyset$ for all $i \in A$, $\alpha \notin \mathrm{dom}(i)$ for all $i \in A$. Therefore, A' and ρ which are introduced in the inference rule of **extend_index** can be defined from A. Finally, if there exist $\overline{\sigma_4}$ and T_4 such that $\overline{C_4}, D, \overline{\sigma}, A' \Downarrow_{tgt} \overline{\sigma_4}, T_4$, we can apply the inference rule of **extend_index** to $\overline{C}$, and it results as follows: $(\textbf{extend\_index}(\alpha, n_+)\ C), D, \overline{\sigma}, A \Downarrow_{tgt} \overline{\sigma_4}\langle\rho\rangle, T_4'$, where $T_4' = [i \mapsto \sum_{0 \le k < n_+} T_4(i +\!\!+ [(\alpha, k)]) : i \in A]$. Thus, what we need to do is to find such $\overline{\sigma_4}$ and T_4, and it can be achieved from the next statement.

(ii) $\hat{C} \equiv (\textbf{loop\_fixpt\_noacc}(n)\{\ \overline{C_3}\ \})$ and $\hat{A} \equiv A'$
There exist two rules for **loop_fixpt_noacc**(n), but premises of both rules can be derived from the common premises as follows:

$$\sigma_0 = \overline{\sigma} \quad \overline{C_3}, D, \sigma_k, A' \Downarrow_{tgt} \sigma_{k+1}, T_{k+1} \text{ for all } k \in \{0, \ldots, n_+ - 1\}$$

If all states are different with each other, it satisfies the second rule, otherwise it satisfies the first rule. Moreover, this common premises can be satisfied from the next statement.

(iii) $\hat{C} \equiv (\textbf{shift}(\alpha);\ \overline{C_2})$ and $\hat{A} \equiv A'$
From the inference rule of **shift**, $\textbf{shift}(\alpha), D, \overline{\sigma}, A' \Downarrow_{tgt} \overline{\sigma}\langle\rho\rangle', T^z_{\overline{A}}$, where ρ is from the premise of the rule. Then, we need to construct a premise for $\overline{C_2}$ for sequential composition, and it follows from the next statement.

(iv) $\hat{C} \equiv (x^{\mathsf{int}\dagger} := \textbf{lookup\_index}(\alpha));\ \overline{C_1}$ and $\hat{A} \equiv A'$
From the definition of A', for any $i' \in A'$, there exist $i \in A, k \in \{0, \ldots, n_+ - 1\}$ such that $i' = i +\!\!+ [(\alpha, k)]$. Therefore, we can apply the rule of **lookup_index** to the first command. Moreover, $\mathcal{S}(\overline{C_1}) \cap \mathrm{dom}(i) \subseteq \mathcal{S}(\overline{C_1}) \cap (\mathrm{dom}(i') \cup \{\alpha\}) = (\mathcal{S}(\overline{C_1}) \cap \mathrm{dom}(i')) \cup (\mathcal{S}(\overline{C_1}) \cap \{\alpha\})$. Then, since $\alpha \notin \mathcal{S}(\overline{C_1})$ and $\mathcal{S}(\overline{C_1}) \subseteq \mathcal{S}(\overline{C})$, $\mathcal{S}(\overline{C_1}) \cap \mathrm{dom}(i) = \emptyset$. Thus, A' satisfies the condition of induction hypothesis for $\overline{C_1}$, and finally we can execute $\overline{C_1}$ by applying the hypothesis.

□

Combining Theorem 2, Proposition 2, Theorem 3, and Lemma 10, we show the following theorem, which describes a sense in which the parallelising translation of C in the source language to $\overline{C}$ in the target language preserves the semantics.

Theorem 4. *Let C be a command in the source language, and $\overline{C}$ be the parallelising translation of C in the target language. Let $D \in RDB$, $\sigma_0 \in State_{src}$, and $\sigma_1 \in State_{tgt}$ such that*

$$\sigma_0 \sim \sigma_1 \quad \textit{and} \quad \sigma_1 \textit{ is a } \{[]\}\textit{-state.}$$

Then, there exist $\sigma_0' \in State_{src}$, $\sigma_1' \in State_{tgt}$, $r \in \mathbb{R}$, and $T \in Tensor_{\mathbb{R}}$ such that

$$C, D, \sigma_0 \Downarrow \sigma_0', r, \qquad \overline{C}, D, \sigma_1, \{[]\} \Downarrow_{tgt} \sigma_1', T, \qquad \sigma_0' \sim \sigma_1', \quad \textit{and} \quad r \sim T.$$

Proof. By Lemma 10, there exist $\sigma_0' \in State_{src}$ and $r \in \mathbb{R}$ such that

$$C, D, \sigma_0 \Downarrow \sigma_0', r.$$

Let C' be the command in the target language obtained from C by replacing all variables $x^{\mathsf{int}\dagger}$ with $x^{\mathsf{int}\dagger\dagger}$. Then, $\overline{C} \equiv \overline{C'}$. By Proposition 2, since $\sigma_0 \sim \sigma_1$, there exist $\sigma_1'' \in State_{tgt}$ and $T' \in Tensor_{\mathbb{R}}$ such that

$$C', D, \sigma_1, \{[]\} \Downarrow_{int} \sigma_1'', T', \qquad \sigma_0' \sim \sigma_1'', \quad \text{and} \quad r \sim T'. \tag{9}$$

Again, by Lemma 10, there exist $\sigma_1' \in State_{tgt}$ and $T \in Tensor_{\mathbb{R}}$ such that

$$\overline{C'}, D, \sigma_1, \{[]\} \Downarrow_{int} \sigma_1', T. \tag{10}$$

Now, consider Theorem 3 where $R = \Delta_{Index}$, and $A_0 = A_1 = \{[]\}$, and $\sigma_0 = \sigma_1$, and $\sigma_0' = \sigma_1''$, and $T_0 = T'$, and $T_1 = T$. Then,Eq. (9) and Eq. (10) imply that

$$\sigma_1'' =_{Index} \sigma_1' \quad \text{and} \quad T' = T.$$

Also, we conclude that $\sigma_0' \sim \sigma_1'$ and $r \sim T$, as desired. Finally, by Theorem 2 and $\overline{C} \equiv \overline{C'}$, Eq. (10) is equivalent to

$$\overline{C}, D, \sigma_1, \{[]\} \Downarrow_{tgt} \sigma_1', T.$$

□

Corollary 5. *Let C be a command in the source language, and $\overline{C}$ be the parallelising translation of C in the target language. Let $D \in RDB$, $\sigma_0 \in State_{src}$, and $\sigma_1 \in State_{tgt}$. If*

$$\sigma_1(x^{\tau\dagger}) = [[] \mapsto \sigma_0(x^\tau)] \quad \textit{for all variables } x^\tau \textit{ in the source language,}$$

then there exist $\sigma_0' \in State_{src}$, $\sigma_1' \in State_{tgt}$, and $r \in \mathbb{R}$ such that

$$C, D, \sigma_0 \Downarrow \sigma_0', r, \quad \overline{C}, D, \sigma_1, \{[]\} \Downarrow_{tgt} \sigma_1', [[] \mapsto r], \quad \textit{and} \quad \sigma_1'(x^{\tau\dagger}) = [[] \mapsto \sigma_0'(x^\tau)] \quad \textit{for all } x^\tau.$$

Proof. This is a direct consequence of Theorem 4 by the definition of $\sigma_0 \sim \sigma_1$ and $r \sim T$ and being a $\{[]\}$-state. □

B Benchmark Structures and Method Applicability

In this section, we describe the structure of the benchmarks and the applicability of each method, as evaluated in §6. Table 3 shows the lengths of the loops in each model, and the averaged number of innermost iterations executed to compute the score when each method is applied. Each method targets different classes of loop structures, which determines its applicability to the benchmarks.

The `seq` method does not apply any vectorisation and executes all iterations sequentially. Although it is applicable to all benchmarks, it performs inefficiently due to the large number of innermost iterations in all benchmarks.

The `plate` method applies *plate constructors* to loop levels that do not contain data dependencies and if-then-else statements. These loop levels are highlighted in light grey in Table 3, and appear in all benchmarks except `arm` and `tcm`.

The `vmarkov` method builds on `plate` by also supporting *vectorised Markov constructors*. These can handle loop levels that may have data dependencies but are not nested and do not include if-then-else branches. These loop levels are highlighted in dark grey in Table 3. As a result, `vmarkov` is applicable to the `hmm-*`, `dmm`, and `arm` benchmarks.

The `discHMM` method combines plate constructors and *discrete HMM constructors*, which can vectorise loop levels corresponding to first-order discrete HMMs, as long as they are not nested. Consequently, `discHMM` is only applicable to the `hmm-ord1` and `hmm-neural` benchmarks.

The `hand` method manually re-writes vectorised code for all loops in the model. This method requires specialised expertise in the internal mechanisms of the specific PPL being used, along with the ability to manipulate tensor-based operations, which can be complex and error-prone (e.g., shape management). For example, in our experiments using Pyro, which does not provide low-level primitives like **fetch** or **score** used in our source and target languages, hand-coded vectorisation necessitated careful manipulation of internal variables used by Pyro's backend algorithms to implement equivalent functionality. We also consider `hand` inapplicable to `tcm`, which contains if-then-else statements where the conditions themselves need to be vectorised, as manually vectorising such conditional constructs is highly non-trivial.

In contrast, `ours` supports vectorisation across all loop structures present in the benchmarks. It can handle data dependencies, nested structures, and if-then-else statements. As a result, `ours` is applicable to all benchmarks and consistently reduces the number of innermost iterations.

	loop lengths	# iter of `seq`	# iter of `plate`	# iter of `vmarkov`	# iter of `discHMM`	# iter of `hand`	# iter of `ours`	inference engines
`hmm-ord1`	229 × 129 × 51	1,506,591	129	3	1	1	2	MAP / MCMC
`hmm-neural`	229 × 129	29,541	129	3	1	1	2	MAP
`hmm-ord2`	50 × 129 × 51	328,950	129	5	-	1	3	MAP / MCMC
`dmm`	229 × 129	29,541	129	3	-	1	2	SVI
`nhmm-train`	5 × 12 × 31 × 24	44,640	8,928	-	-	1	8	MAP / MCMC
`nhmm-stock`	10 × 10 × 4 × 64	25,600	2,560	-	-	1	8	MAP / MCMC
`arm`	2000	2,000	-	19.2	-	1	10.1	SVI / MCMC
`tcm`	10 × 200	2,000	-	-	-	-	2	SVI / MCMC

Table 3. Lengths of loops in each benchmark, and averaged number of innermost iterations executed to compute the score for each method, and the inference engines used in the experiments. The averaged number of iterations for `arm` may not be an integer due to the randomly sampled order. The loop levels highlighted in light grey are applicable to plate constructors, and those in dark grey are applicable to vectorised Markov constructors. "-" indicates that the corresponding method is not applicable to the benchmark.

C Code Snippet of Benchmarks

In this section, we provide code snippets of the `hmm-neural` benchmark using different vectorisation strategies. Some components of the code (e.g., library imports, neural network definitions) are omitted for brevity. The complete implementation of the benchmark is available at https://github.com/Lim-Sangho/auto-vectorise-ppl.public.

Fig. 13 illustrates the baseline implementation without vectorisation (`seq`). This version defines a hidden Markov model (HMM) with neural network emissions (`tones_generator`), which observes sequences of piano tones (`sequences`). It uses two nested loops: the outer loop iterates over sequences of music in a batch, and the inner loop iterates over the time steps within each sequence. Each `sample` statement reads the values from the random database using variable names formatted as `f"x_{i}_{t}"` and `f"y_{i}_{t}"`, where `i` and `t` denote the sequence and time step indices, respectively. The first `sample` statement includes the argument `infer = {"enumerate": "parallel"}`, which instructs Pyro to perform parallel enumeration and variable elimination over the discrete latent variable.

Fig. 14 shows our vectorised immplementation (`ours`), which employs the `@vec.vectorize` decorator and the `vec.range` constructor to parallelise the loops while preserving the original sequential structure. This version requires an additional argument, `s` of type `State`, to the model for managing variable read and write operations, and an `Index` operator to support nan-mask based indexing, where nan values represent empty indices in a partial map in the state. These additions are orthogonal to the model structure and independent of tensor shapes or dimensionality, allowing users to mitigate manual tensor manipulations typically required in many existing vectorisation approaches.

Figs. 15 and 16 present the implementations using the *plate constructors* (`plate`), and the combination of *plate* and *vectorised Markov constructors* (`vmarkov`), respectively. These versions require explicit tensor shape manipulations (e.g., through slice indexing and `unsqueeze`) to align with the semantics of the respective constructors. Also, these strategies are only valid when loop iterations are conditionally independent or when data dependencies are explicitly specified by the user (e.g., via the `history` argument in `vectorized_markov`). In the `hmm-neural` benchmark, the outer loop can be vectorised using `plate`, while the inner loop requires `vectorized_markov` with an argument `history=1` due to the dependencies on previous time steps.

Fig. 17 shows the version implemented using *plate constructors* in combination with *discrete HMM constructors* (`discHMM`). This version leverages Pyro's `DiscreteHMM` distribution, which is designed for discrete hidden Markov models. It requires substantial restructuring of the original sequential code, including explicit tensor shape management through operations such as `unsqueeze` and `cat`.

Finally, Fig. 18 presents hand-vectorised implementation (`hand`). Similar to the previous case, this version requires extensive manual modifications, such as giving a non-standard argument (i.e., `infer={"_do_not_trace": True, "is_auxiliary": True}`) to the `sample` statement to suppress errors caused by duplicate sampling variable names, and explicitly handling tensor shapes through operations such as `unsqueeze`, `cat`, and `repeat`.

```
def hmm_neural_seq(sequences, lengths, tones_generator, hidden_dim=16):
    num_sequences, max_length, data_dim = sequences.shape
    probs_x = sample(
        "probs_x",
        Dirichlet(0.9 * eye(hidden_dim) + 0.1).to_event(1),
    )

    for i in range(num_sequences):
        x = 0
        y = zeros(data_dim)
        for t in range(max_length):
            with mask(mask=(t < lengths[i])):
                x = sample(
                    f"x_{i}_{t}",
                    Categorical(logits=probs_x[x].log()),
                    infer={"enumerate": "parallel"},
                )
                y = sample(
                    f"y_{i}_{t}",
                    Bernoulli(logits=tones_generator(x, y)).to_event(1),
                    obs=sequences[i, t],
                )
```

Fig. 13. A code snippet of the hmm-neural benchmark with no vectorisation (seq).

```
@vec.vectorize
def hmm_neural_ours(s: State, sequences, lengths, tones_generator, hidden_dim=16):
    num_sequences, max_length, data_dim = sequences.shape
    probs_x = sample(
        "probs_x",
        Dirichlet(0.9 * eye(hidden_dim) + 0.1).to_event(1),
    )

    for i in vec.range("sequences", num_sequences, vectorized=True):
        s.x = 0
        s.y = zeros(data_dim)
        for t in vec.range("lengths", max_length, vectorized=True):
            with mask(mask=(t < lengths[i])):
                s.x = sample(
                    "x",
                    Categorical(logits=Index(probs_x)[s.x].log()),
                    infer={"enumerate": "parallel"},
                )
                s.y = sample(
                    "y",
                    Bernoulli(logits=tones_generator(s.x, s.y)).to_event(1),
                    obs=Index(sequences)[i, t],
                )
```

Fig. 14. A code snippet of the hmm-neural benchmark implemented with our proposed vectorisation constructors (ours).

```
def hmm_neural_plate(sequences, lengths, tones_generator, hidden_dim=16):
    num_sequences, max_length, data_dim = sequences.shape
    probs_x = sample(
        "probs_x",
        Dirichlet(0.9 * eye(hidden_dim) + 0.1).to_event(1),
    )

    with plate("sequences", num_sequences):
        x = 0
        y = zeros(data_dim)
        for t in range(max_length):
            with mask(mask=(t < lengths)):
                x = sample(
                    f"x_{t}",
                    Categorical(logits=probs_x[x].log()),
                    infer={"enumerate": "parallel"},
                )
                y = sample(
                    f"y_{t}",
                    Bernoulli(logits=tones_generator(x, y)).to_event(1),
                    obs=sequences[:, t],
                )
```

Fig. 15. A code snippet of the hmm-neural benchmark implemented with *plate constructors* (plate).

```
def hmm_neural_vmarkov(sequences, lengths, tones_generator, hidden_dim=16):
    num_sequences, max_length, data_dim = sequences.shape
    probs_x = sample(
        "probs_x",
        Dirichlet(0.9 * eye(hidden_dim) + 0.1).to_event(1),
    )

    with plate("sequences", num_sequences, dim=-2) as batch:
        batch = batch.unsqueeze(-1)
        x = 0
        y = zeros(data_dim)
        for t in vectorized_markov(None, "lengths",
                                   max_length, dim=-1, history=1):
            with mask(mask=(t < lengths.unsqueeze(-1))):
                x = sample(
                    f"x_{t}",
                    Categorical(logits=probs_x[x].log()),
                    infer={"enumerate": "parallel"},
                )
                y = sample(
                    f"y_{t}",
                    Bernoulli(logits=tones_generator(x, y)).to_event(1),
                    obs=sequences[batch, t],
                )
```

Fig. 16. A code snippet of the hmm-neural benchmark implemented with *plate constructors* and *vectorised Markov constructors* (vmarkov).

```
def hmm_neural_discHMM(sequences, lengths, tones_generator, hidden_dim=16):
    num_sequences, max_length, data_dim = sequences.shape
    probs_x = sample(
        "probs_x",
        Dirichlet(0.9 * eye(hidden_dim) + 0.1).to_event(1),
    )

    with plate("sequences", num_sequences):
        mask = (arange(max_length) < lengths.unsqueeze(-1))
        x = arange(hidden_dim)
        y = cat([zeros(num_sequences, 1, data_dim), sequences[:, :-1]], dim=1)
        init_logits = cat([ones(1), zeros(hidden_dim - 1)]).log()
        trans_logits = where(mask.unsqueeze(-1).unsqueeze(-1), probs_x.log(), 0)
        obs_dist = Bernoulli(
            logits=tones_generator(x, y.unsqueeze(-2))
        ).to_event(1)
        obs_dist = obs_dist.mask(mask.unsqueeze(-1))
        hmm_dist = DiscreteHMM(init_logits, trans_logits, obs_dist)
        sample("y", hmm_dist, obs=sequences)
```

Fig. 17. A code snippet of the `hmm-neural` benchmark implemented with *plate constructors* and *discrete HMM constructors* (`discHMM`).

```
def hmm_neural_hand(sequences, lengths, tones_generator, hidden_dim=16):
    num_sequences, max_length, data_dim = sequences.shape
    probs_x = sample(
        "probs_x",
        Dirichlet(0.9 * eye(hidden_dim) + 0.1).to_event(1)
    )

    with mask(mask=(arange(max_length) < lengths.unsqueeze(-1))):
        x_prev = sample(
            "x",
            Categorical(logits=zeros(hidden_dim)),
            infer={"enumerate": "parallel",
                   "_do_not_trace": True,
                   "is_auxiliary": True}
        )
        x_prev = cat([zeros(1),
                      x_prev.repeat(1, num_sequences, max_length-1)],
                      dim=-1)

        x_curr = sample(
            "x",
            Categorical(logits=probs_x[x_prev].log()),
            infer={"enumerate": "parallel"}
        )

        y_prev = cat([zeros(num_sequences, 1, data_dim),
                      sequences[:, :-1]],
                      dim=-2)

        sample(
            "y",
            Bernoulli(logits=tones_generator(x_curr, y_prev)).to_event(1),
            obs=sequences
        )
```

Fig. 18. A code snippet of the hmm-neural benchmark implemented with hand-coded vectorisation (hand).

```
def MAP_step(model, optim, sequence, lengths, tones_generator):
    model_trace = trace(model).get_trace(sequence, lengths, tones_generator)
    log_prob = model_trace.compute_log_prob()
    params = model_trace.get_params()
    optim(log_prob, params)  # Update params to maximise log_prob

optim = Adam({"lr": 1e-1})
sequence, lengths = Data()
tones_generator = TonesGeneratorNN()
model = hmm_neural_ours  # or hmm_neural_seq, hmm_neural_plate, etc.

for step in range(num_steps):
    MAP_step(model, optim, sequence, lengths, tones_generator)
```

Fig. 19. A code snippet of a training step for MAP inference used in all versions of the hmm-neural benchmark.

D Additional Results from Experimental Evaluation

In this section, we provide detailed results from the experimental evaluation on RQ2 in §6.

- Table 4 shows the averaged time to compute the sum of scores (score time) and to run the single entire training step (total time) of each model in SVI or MAP inference.
- Table 5 shows the maximum GPU memory usage during the execution of each training step in SVI or MAP inference.
- Table 6 shows the averaged time to execute each MCMC step for each model.
- Table 7 shows the maximum GPU memory usage during the execution of each MCMC step.

All tables also include the ratios over `ours` and standard deviations.

Our results show that reducing the number of innermost iterations leads to greater time savings. For example, when comparing `ours` with `plate`, in the case of `hmm-ord2`, `plate` executes 129 innermost iterations, whereas `ours` reduces this to just 3, achieving a 43× reduction in iteration count, which results in 4.5× speedup in score time and 1.8× speedup in total time during SVI or MAP inference, and 1.8× speedup in time per MCMC step. In contrast, for `nhmm-train`, the number of innermost iterations decreases from 8,928 (`plate`) to 8 (`ours`), resulting in a reduction of over 1,116×, leading to significantly greater time savings: 145.8× speedup in score time and 78.4× speedup in total time during SVI or MAP inference, and 65.6× speedup in time per MCMC step.

Additionally, we observe that `ours` achieves memory reductions in some benchmarks compared to `seq`, especially in those with relatively low memory demands, such as `nhmm-stock`, `arm`, and `tcm`. This is because, in these benchmarks, the `seq` method often operates on small-sized vectors within each innermost iteration. When the GPU allocates memory for these vectors, it tends to allocate many small chunks, which increases memory overhead. In contrast, `ours` allocates larger memory chunks at once due to its vectorised structure, resulting in lower memory overhead.

	SVI/MAP score time (s) (↓)						SVI/MAP total time (s) (↓)					
	seq	plate	vmarkov	discHMM	hand	ours	seq	plate	vmarkov	discHMM	hand	ours
hmm-ord1	669.097	0.118	0.054	0.006	0.006	0.021	1066.060	0.192	0.083	0.018	0.035	0.075
hmm-neural	35.715	0.158	0.056	0.010	0.009	0.028	60.474	0.275	0.082	0.027	0.042	0.062
hmm-ord2	131.594	0.124	0.165	-	0.009	0.027	205.457	0.282	0.280	-	0.127	0.158
dmm	42.092	0.199	0.076	-	0.004	0.016	129.339	0.957	0.524	-	0.427	0.496
nhmm-train	91.287	18.523	-	-	0.015	0.127	121.993	25.021	-	-	0.198	0.319
nhmm-stock	29.925	3.058	-	-	0.005	0.089	57.417	6.247	-	-	0.022	0.139
arm	1.129	-	0.177	-	0.003	0.031	2.354	-	0.199	-	0.009	0.043
tcm	1.653	-	-	-	-	0.029	3.837	-	-	-	-	0.045
	SVI/MAP score time ratio (· / ours) (↓)						SVI/MAP total time ratio (· / ours) (↓)					
	seq	plate	vmarkov	discHMM	hand	ours	seq	plate	vmarkov	discHMM	hand	ours
hmm-ord1	31399.3	5.6	2.6	0.3	0.3	1	14302.9	2.6	1.1	0.2	0.5	1
hmm-neural	1291.6	5.7	2.0	0.4	0.3	1	972.4	4.4	1.3	0.4	0.7	1
hmm-ord2	4824.4	4.5	6.0	-	0.3	1	1300.4	1.8	1.8	-	0.8	1
dmm	2657.3	12.6	4.8	-	0.3	1	260.7	1.9	1.1	-	0.9	1
nhmm-train	718.5	145.8	-	-	0.1	1	382.2	78.4	-	-	0.6	1
nhmm-stock	337.0	34.4	-	-	0.1	1	414.1	45.1	-	-	0.2	1
arm	36.2	-	5.7	-	0.1	1	54.8	-	4.6	-	0.2	1
tcm	57.7	-	-	-	-	1	85.6	-	-	-	-	1
	SVI/MAP score time std. (s)						SVI/MAP total time std. (s)					
	seq	plate	vmarkov	discHMM	hand	ours	seq	plate	vmarkov	discHMM	hand	ours
hmm-ord1	53.727	0.018	0.004	0.000	0.000	0.001	50.168	0.018	0.005	0.001	0.001	0.002
hmm-neural	0.387	0.005	0.004	0.000	0.000	0.001	3.242	0.019	0.005	0.001	0.001	0.002
hmm-ord2	1.919	0.019	0.008	-	0.000	0.000	2.344	0.021	0.010	-	0.008	0.005
dmm	0.555	0.031	0.002	-	0.000	0.000	9.477	0.046	0.009	-	0.004	0.005
nhmm-train	1.036	0.223	-	-	0.000	0.001	1.132	0.234	-	-	0.003	0.003
nhmm-stock	2.025	0.056	-	-	0.000	0.001	2.498	0.076	-	-	0.002	0.002
arm	0.064	-	0.065	-	0.001	0.010	0.143	-	0.066	-	0.001	0.010
tcm	0.064	-	-	-	-	0.000	0.220	-	-	-	-	0.000

Table 4. Averaged score time (s) and total time (s) for each training step, their ratios over ours, and standard deviations in SVI or MAP inference. We report the average over 5 runs × 9 training steps for seq, and 5 runs × 990 training steps for other methods. Lower is better.

	SVI/MAP GPU memory usage (GB) (↓)					
	seq	plate	vmarkov	discHMM	hand	ours
hmm-ord1	5.053	0.700	1.582	0.277	0.888	1.774
hmm-neural	0.723	0.518	1.300	0.483	1.002	1.043
hmm-ord2	1.902	0.868	5.229	-	2.289	2.467
dmm	0.871	0.622	1.201	-	0.679	0.739
nhmm-train	3.342	2.851	-	-	3.899	3.989
nhmm-stock	0.260	0.057	-	-	0.039	0.040
arm	0.015	-	0.011	-	0.000	0.002
tcm	0.022	-	-	-	-	0.001
	SVI/MAP GPU memory usage ratio (· / ours) (↓)					
	seq	plate	vmarkov	discHMM	hand	ours
hmm-ord1	2.849	0.395	0.892	0.156	0.501	1
hmm-neural	0.693	0.496	1.246	0.463	0.961	1
hmm-ord2	0.771	0.352	2.120	-	0.928	1
dmm	1.179	0.841	1.626	-	0.919	1
nhmm-train	0.838	0.715	-	-	0.977	1
nhmm-stock	6.487	1.420	-	-	0.963	1
arm	9.842	-	7.313	-	0.193	1
tcm	28.872	-	-	-	-	1
	SVI/MAP GPU memory usage (GB) std.					
	seq	plate	vmarkov	discHMM	hand	ours
hmm-ord1	0.000	0.000	0.000	0.000	0.000	0.000
hmm-neural	0.000	0.000	0.000	0.000	0.000	0.000
hmm-ord2	0.000	0.000	0.000	-	0.000	0.000
dmm	0.006	0.001	0.001	-	0.001	0.001
nhmm-train	0.003	0.000	-	-	0.000	0.000
nhmm-stock	0.001	0.000	-	-	0.000	0.000
arm	0.000	-	0.001	-	0.000	0.000
tcm	0.000	-	-	-	-	0.000

Table 5. Averaged GPU memory usage (GB) for each training step, its ratio over ours, and standard deviation in SVI or MAP inference. We report the average over 5 runs × 9 training steps for seq, and 5 runs × 990 training steps for other methods. Lower is better.

	MCMC time per step (s) (↓)					
	seq	plate	vmarkov	discHMM	hand	ours
hmm-ord1	nan	1.872	0.844	0.223	0.399	0.786
hmm-ord2	1868.995	2.826	2.734	-	1.238	1.545
nhmm-train	1232.830	245.348	-	-	2.512	3.738
nhmm-stock	510.519	52.833	-	-	0.560	1.439
arm	21.952	-	0.489	-	0.049	0.191
tcm	28.087	-	-	-	-	0.417
	MCMC time per step ratio (· / ours) (↓)					
	seq	plate	vmarkov	discHMM	hand	ours
hmm-ord1	nan	2.383	1.074	0.284	0.507	1
hmm-ord2	1209.649	1.829	1.769	-	0.801	1
nhmm-train	329.769	65.628	-	-	0.672	1
nhmm-stock	354.730	36.710	-	-	0.389	1
arm	114.806	-	2.557	-	0.256	1
tcm	67.277	-	-	-	-	1
	MCMC time per step (s) std.					
	seq	plate	vmarkov	discHMM	hand	ours
hmm-ord1	nan	0.093	0.020	0.016	0.020	0.027
hmm-ord2	84.462	0.176	0.100	-	0.067	0.096
nhmm-train	38.036	6.592	-	-	0.036	0.081
nhmm-stock	30.262	2.965	-	-	0.035	0.078
arm	0.902	-	0.069	-	0.004	0.018
tcm	0.859	-	-	-	-	0.027

Table 6. Average time per MCMC step (s) in 1 hour, its ratio over ours, and standard deviation. Each reported number is averaged over 8 runs × averaged number of steps executed in each one-hour run. A dash (-) indicates that the method is not applicable to the benchmark. A value of nan indicates it fails to execute a single MCMC step within 1 hour. Lower is better.

	MCMC GPU memory usage (GB) (↓)					
	seq	plate	vmarkov	discHMM	hand	ours
hmm-ord1	nan	0.7000	1.2663	0.2464	0.6681	1.3834
hmm-ord2	1.8299	0.8659	3.8962	-	2.0342	2.1895
nhmm-train	3.3428	2.8506	-	-	3.6922	3.7635
nhmm-stock	0.2446	0.0517	-	-	0.0344	0.0358
arm	0.0103	-	0.0090	-	0.0002	0.0022
tcm	0.0139	-	-	-	-	0.0005
	MCMC GPU memory usage (GB) ratio (· / ours) (↓)					
	seq	plate	vmarkov	discHMM	manual	ours
hmm-ord1	nan	0.5060	0.9153	0.1781	0.4830	1
hmm-ord2	0.8358	0.3955	1.7795	-	0.9291	1
nhmm-train	0.8882	0.7574	-	-	0.9811	1
nhmm-stock	6.8400	1.4457	-	-	0.9607	1
arm	4.7768	-	4.1773	-	0.0775	1
tcm	29.5086	-	-	-	-	1
	MCMC GPU memory usage (GB) std.					
	seq	plate	vmarkov	discHMM	manual	ours
hmm-ord1	nan	0.0000	0.0000	0.0000	0.0000	0.0000
hmm-ord2	0.0000	0.0000	0.0000	-	0.0000	0.0000
nhmm-train	0.0000	0.0000	-	-	0.0000	0.0000
nhmm-stock	0.0000	0.0000	-	-	0.0000	0.0000
arm	0.0000	-	0.0001	-	0.0000	0.0010
tcm	0.0000	-	-	-	-	0.0000

Table 7. Average GPU memory usage (GB) for each MCMC step in 1 hour, its ratio over ours, and standard deviation. Each reported number is averaged over 8 runs × averaged number of steps executed in each one-hour run. A value of nan indicates it fails to execute a single MCMC step within 1 hour. A dash (-) indicates that the method is not applicable to the benchmark. Lower is better.