Gradual Tensor Shape Checking (full version)

Momoko Hattori, Naoki Kobayashi, and Ryosuke Sato

The University of Tokyo, Tokyo, Japan
{momohatt,koba,rsato}@is.s.u-tokyo.ac.jp

Abstract. Tensor shape mismatch is a common source of bugs in deep learning programs. We propose a new type-based approach to detect tensor shape mismatches. One of the main features of our approach is the best-effort shape inference. As the tensor shape inference problem is undecidable in general, we allow static type/shape inference to be performed only in a best-effort manner. If the static inference cannot guarantee the absence of the shape inconsistencies, dynamic checks are inserted into the program. Another main feature is gradual typing, where users can improve the precision of the inference by adding appropriate type annotations to the program. We formalize our approach and prove that it satisfies the criteria of gradual typing proposed by Siek et al. in 2015. We have implemented a prototype shape checking tool based on our approach and evaluated its effectiveness by applying it to some deep neural network programs.

1 Introduction

Tensor Shape Checking and Its Difficulties. Tensor shape mismatch is one of the common sources of dynamic errors in programs using tensors (i.e., multi-dimensional arrays). For example, the reshape operation of tensors takes a tensor \( x \) and an integer list \( S \) and returns a new tensor of the shape \( S \) obtained by realigning the elements in \( x \). The input and output tensors must have the same number of elements; a tensor of shape \([2; 3; 4]\)\(^1\) can be reshaped into a shape \([3; 2; 4]\), while trying to reshape it into \([3; 4]\) results in a runtime error.

Early detection of tensor shape mismatch errors is critical in particular for deep learning programs, where tensors are frequently used. Since deep learning programs often take a considerable amount of time to train networks, it is often the case that a program takes hours and days to compute the weights of deep neural networks only to be terminated by one tensor shape mismatch error, throwing away the trained weights. Even worse, some tensor shape mismatches can be harder to notice: mixing up the height and the width of square images does not raise runtime errors but degrades the performance of the neural network.

The existing work on static detection of tensor shape mismatch errors can be classified into two categories. One is the whole-program analysis approach \([16,30]\), which collects tensor shape information by partially evaluating

\(^1\) In this paper, we denote lists in the OCaml-style as in \([1; 2; 3]\) to disambiguate it from the citations.
let model s =
let f = ... in let g = ... in fun x -> let y = f x in g y

let _ = model 1 (Tensor.rand [20])

Fig. 1. An OCaml program written with OCaml-Torch.

the program in the style of abstract interpretation. The other is the type-based approach [3,24], which expresses the shapes of tensors as a part of the type information. Still, none of them is fully satisfactory: either they are too conservative and reject valid programs, or fail to detect some shape mismatch errors.

This paper pursues the type-based approach as it is expected to provide modular detection of tensor shape inconsistencies. Designing an appropriate type system and a type inference procedure to reason about tensor shapes is challenging because shapes are first-class objects. For example, the library function Tensor.zeros of OCaml-Torch [4] (which provides OCaml bindings for libtorch [19]) takes a list $S$ of integers, and returns a new tensor whose shape is $S$. Thus, we have to work with dependent types: Tensor.zeros would be given the type $S: \text{int list} \rightarrow \{ r: \text{tensor} | r.\text{shape} = S \}$. It is difficult to infer such dependent (refinement) types fully automatically. Yet, we wish to avoid programmers’ burden of writing too many type annotations.

Another difficulty is that shape constraints can be so complex that even type checking, let alone inference, can be too costly or impossible. For instance, the reshape operation explained earlier needs the proof that the shape of the input tensor $x$ is compatible with the given shape $S = [s_1; \ldots ; s_n]$ (i.e., if the shape of $x$ is to be $[s'_1; \ldots ; s'_m]$, then $\prod_{i=1}^m s'_i = \prod_{i=1}^n s_i$ holds). Thus, type checking requires complex reasoning about (non-linear) integer arithmetic and lists.

Overview of Our Approach. Based on the observations above, we propose an approach that is expected to work well in practice despite the above-mentioned difficulties. Our approach can be characterized by three main features: best-effort type inference, hybrid type checking, and gradual typing [26]. We explain them using our prototype tool GRATEN$^3$.

Best-Effort Type Inference. GRATEN does not try to infer the most general types; it performs type/shape inference in a best-effort manner. Thanks to this design choice, GRATEN works even if no type annotations are provided (despite that the underlying type system involves dependent types), and yet it can statically detect (not necessarily all but) some shape mismatch errors.

As an example, let us consider the program in Figure 1. The function model takes an integer parameter $s$, defines functions $f$ and $g$, and returns a layer (which is a function that takes a tensor and returns a tensor) which composes $f$

\[2 \text{ Actually, some } s_i \text{ can be } -1, \text{ in which case the size of the } i-\text{th dimension is unspecified.}\]

\[3 \text{ The tool is publicly available at https://doi.org/10.5281/zenodo.7590480. The source code is also publicly available at https://github.com/momohatt/graten.}\]
and \( g \). The definitions of \( f \) and \( g \) are omitted here, but their types are assumed as below, where \( s \) in the type of \( f \) is the argument of \( s \) and the function \( \text{nth}(n, S) \) returns the \( n \)-th element of the list \( S \) (the index starts with 0).

\[
\begin{align*}
\text{f} : & \{ \nu : \text{tensor} \mid \text{len}(\nu.\text{shape}) = 1 \} \rightarrow \text{tensor}(\text{nth}(0, \nu.\text{shape})/s) \\
\text{g} : & \text{tensor}([10]) \rightarrow \text{tensor}([1])
\end{align*}
\]

These types indicate that \( f \) takes a 1-dimensional tensor (i.e., a vector) and returns a vector whose length equals the length of the argument vector divided by \( s \), and that \( g \) expects a vector of length 10 and returns a vector of length 1.

The formal syntax of types will be introduced later in Section 2.

For the program above, \textsc{Graten}'s best-effort inference outputs the following type for the function \( \text{model} \).

\[
\begin{align*}
s : \text{int} \rightarrow & : \{ \nu : \text{tensor} \mid \text{len}(\nu.\text{shape}) = 1 \land \text{nth}(0, \nu.\text{shape})/s = 10 \} \rightarrow \text{tensor}([1])
\end{align*}
\]

Hybrid Type Checking. Another main feature of our approach is hybrid type checking: we combine static and dynamic checking. The type checker inserts assertions to program points where the type safety is not statically guaranteed, à la Knowles and Flanagan’s hybrid type checking [15]. For example, consider the program in Figure 2, which is obtained by adding a conditional branch to the one in Figure 1. The type of the \textit{then} and \textit{else} branch of the if expression are inferred to be \( \text{tensor}(x.\text{shape}) \) and \( \text{tensor}([\text{nth}(0, x.\text{shape})/s]) \), respectively. In this case, the type of \( y \) is simply inferred to be \( \text{tensor} \) without any information about its shape, and the inferred type for \( \text{model} \) is as follows.

\[
\begin{align*}
s : \text{int} \rightarrow & : \{ \nu : \text{tensor} \mid \text{len}(\nu.\text{shape}) = 1 \} \rightarrow \text{tensor}([1])
\end{align*}
\]

Thus, the best-effort inference of \textsc{Graten} fails to capture the constraint \( \text{nth}(0, x.\text{shape})/s = 10 \) for \( x \) due to the imprecise type information of \( y \). Along with
let model s =
let f = ... in let g = ... in
fun x ->
let y = ((if s = 1 then x else f x) : tensor([nth 0 x.shape / s]))
in g y

Fig. 4. The program from Figure 2 after adding type annotations.

the inferred types, GraTen outputs the program in Figure 3, which is the same as the original program except for the assertion inserted at the argument of \( g \). Since the statically inferred type of \( y \) fails to guarantee that the application of \( g \) to \( y \) does not lead to a shape mismatch error, GraTen inserts the assertion to check the requirement dynamically.

**Gradual Typing.** Lastly, our approach incorporates gradual typing \([26]\)\(^4\) so that the users can improve the precision of inferred types by adding type annotations. For example, let us consider the program in Figure 4, which is obtained from the one in Figure 2 by adding a type annotation to \( y \). With this annotation, GraTen infers the same type for \( \text{model} \) as it did for \( \text{model} \) in Figure 1, and no assertions are inserted. As such, adding correct type annotations improves the type checking and decreases the number of assertions inserted.

Thanks to the best-effort inference, users need not add type annotations to everywhere in the program. They can focus on the program points where the static inference did not perform well, which is indicated by the insertion of assertions. We prove that our type system satisfies the gradual guarantee \([26]\), which ensures that adding type annotation preserves the type-ability and the behavior of the program (with some assertions inserted) regardless of its precision, as long as the annotation does not disagree with the program.

Among the three features, the notion of hybrid type checking was first proposed by Knowles and Flanagan \([15]\), and our gradual typing is closely related to gradual refinement types by \([17]\), but we believe that the particular combination of three features is new. In particular, unlike the original gradual refinement types \([17]\), we insert assertions instead of carrying around evidence terms \([11]\) in the reduction to guarantee type safety.

The contributions are summarized as follows. (i) The formalization of a type system that combines hybrid type checking and gradual typing. We define our type system as the type-based transformation relation from source programs to programs with run-time assertion checks. We prove the soundness of our type system as well (Section 2). The shape-polymorphic extension of the type system is also briefly discussed (Appendix A). (ii) A proof that our system satisfies the

---

\(^4\) Usually, gradual typing introduces new syntax for gradual types and makes a distinction between static types and gradual types. However, our type system does not have such distinction; it only uses the standard refinement types. As we see later, we extend the standard refinement type system with cast (assertion) insertion rules so that it can be viewed as a gradualized type system.
A Gradually-Typed Language with Refinement Types

In this section, we formalize our type system and the translation to insert assertions. We first introduce the source and target languages of the translation in Sections 2.1 and 2.2. We then formalize the type system and the translation and prove their soundness in Section 2.3. The gradual guarantee is discussed later in Section 3.

2.1 Source Language

We consider a call-by-value functional language, whose syntax is given in Figure 5. Throughout this paper, $n$, $c$, and $x$ respectively denote integers, constants (including integers and primitive functions) and variables. The base types $B$ and refinement predicates $\varphi$ are explained later.

Type annotations can be added to the function arguments $\lambda x: \tau.M$, recursive functions $\text{fix}(f:(x: \tau_1 \rightarrow \tau_2), x, M)$ and to arbitrary expressions by $(M : \tau)$. In the implementation of GraTen, users may omit the type annotations in lambda expressions and recursive functions as the best-effort type inference tries to complete them.

The argument of a function application and the branching condition of an if-expression are restricted to variables for the sake of simplicity of typing rules. Note that this restriction does not lose generality, as a general function application $M_1 M_2$ can be normalized to $\text{let } f = M_1 \text{ in let } x = M_2 \text{ in } f x$.

Types are defined following the standard definition of refinement types. Intuitively, the type $\{x:B \mid \varphi\}$ describes a value $x$ of type $B$ such that $\varphi$ holds. For example, $\{x: \text{int} \mid x \geq 0\}$ is the type of non-negative ints. We may omit the refinement predicates when they are $\text{true}$. For example, we may write $\{x: \text{int} \mid \text{true}\}$ as $\text{int}$.

The language presented so far is general; in GraTen it is instantiated to a language for tensor programs by defining the base types and refinement predicates as in Figure 6, and assuming that primitive operations on tensors are...
\[ B \text{ (base type)} := \text{bool} \mid \text{int} \mid \text{int list} \mid \text{tensor} \]

\[ \varphi \text{ (predicate)} := \text{true} \mid \text{false} \mid s_1 = s_2 \mid S_1 = S_2 \mid x \mid \neg \varphi \mid \varphi_1 \land \varphi_2 \mid \varphi_1 \lor \varphi_2 \mid \text{broadcastable}(S_1, S_2) \mid \text{reshapeable}(S_1, S_2) \]

\[ S \text{ (shape)} := [s_1; \ldots; s_n] \mid x \mid x.\text{shape} \mid \text{cons}(s, S) \mid \text{append}(S_1, S_2) \mid \text{tail}(S) \mid \text{init}(S) \mid \text{insertAt}(s_1, s_2, S) \mid \text{dropAt}(s, S) \mid \text{swap}(s_1, s_2, S) \mid \text{reshape}(S_1, S_2) \mid \text{broadcast}(S_1, S_2) \mid \text{matmul}(S_1, S_2) \]

\[ s \text{ (size)} := n \mid x \mid -s \mid s_1 + s_2 \mid s_1 \times s_2 \mid \frac{s_1}{s_2} \mid \text{head}(S) \mid \text{last}(S) \mid \text{len}(S) \mid \text{nth}(s, S) \mid \text{prod}(S) \]

Fig. 6. Syntax of base types \( B \) and predicates \( \varphi \) in GraTen.

\[ v \text{ (value)} := c \mid x \mid [v_1, \ldots, v_n] \mid \lambda x. N \mid \text{fix}(f^*, x, N) \]

\[ N \text{ (cast term)} := v \mid \text{if } v \text{ then } N_1 \text{ else } N_2 \mid N \upsilon \mid \text{let } x = N_1 \text{ in } N_2 \mid \text{assert}(\varphi); N \]

Fig. 7. Syntax of the target language.

included in the set of constants ranged over \( c \). The refinement predicates, shapes and sizes are expressions of type bool, int list and int respectively. The supported predicates are those described by quantifier-free formulas of first-order logic. As shown in the definition, they may use some built-in predicates and functions over integer lists such as append and primitives on integer arithmetic in order to express common tensor operations. We implicitly assume that the refinement predicates are well formed (as defined in Figure 15, Appendix B).

2.2 Target Language

As explained in Section 1, we insert run-time checks into places where type-safety cannot be statically guaranteed. Figure 7 shows the syntax of programs obtained by the insertion of assertions. A main difference from the source language is the addition of assertion \text{assert}(\varphi); N, which is used to implement the run-time checks. Like Flanagan’s hybrid type system [15] (and unlike the blame calculus [31]), we guarantee the safety of target programs by assertions. Compared with the blame calculus, this method is expected to be easier to implement since most of the modern programming languages are equipped with assertions, and more efficient in that it avoids the accumulation of dynamic casts at runtime. This implementation of the dynamic cast is possible since our system is only “gradualized” at the predicate level of the refinement type and the underlying simple type is static.

Another difference is that the binders in \text{let} expressions are annotated with their type. This is required when defining the precision relation over the cast terms in Section 3.

The substitution and the reduction rules of the cast terms are presented in Figure 8. The evaluation of primitive function \text{ev}(c, v) is defined to be the
Fig. 8. Selected rules of substitution and reduction of the target language (the full definition is given in Figures 16 and 17, Appendix B).

\[
\begin{align*}
[v/x]N & \quad \text{(assert)}(\varphi); N = \text{assert}([v/x]\varphi); [v/x]N \\
\lambda y^\tau.v^\sigma & \quad \text{assert}(\text{true}); N \rightarrow N \\
\lambda y^\tau.v^\sigma & \quad \text{assert}(\text{false}); N \rightarrow \text{error} \\
\end{align*}
\]

Variables are assumed to be alpha-renamed so that variables at different scopes do not collide.

Fig. 9. Typing rules for the cast terms \(\Gamma; \varphi \vdash N : \tau\).

\[
\begin{align*}
\Gamma; \varphi \vdash c : ty(c) \quad & (\text{CT-CON}) \\
\Gamma(x) = y : \tau_1 \rightarrow \tau_2 & (\text{CT-VF}) \\
\Gamma; \varphi \vdash x : \Gamma(x) & (\text{CT-APP})\\
\Gamma, x : \tau_1; \varphi \vdash N : \tau_2 & (\text{CT-LAM}) \\
\Gamma; \varphi \vdash \lambda x^\tau.N : x : \tau_1 \rightarrow \tau_2 & (\text{CT-APPLAM}) \\
\Gamma, f : (x : \tau_1 \rightarrow \tau_2), x : \tau_1; \varphi \vdash N : \tau_2 & (\text{CT-FIX}) \\
\Gamma; \varphi \vdash \text{fix}(f^{x \rightarrow \tau_1 \rightarrow \tau_2}, x, N) : x : \tau_1 \rightarrow \tau_2 & (\text{CT-FIX}) \\
\Gamma; \varphi \vdash v : \{x : \text{bool} | \varphi'\} & (\text{CT-IF}) \\
\Gamma; \varphi \vdash \text{if } v \text{ then } N_1 \text{ else } N_2 : \tau & (\text{CT-IF}) \\
\Gamma; \varphi \vdash N_1 : \tau_1 & (\text{CT-LET}) \\
\Gamma, x : \tau_1; \varphi \vdash N_2 : \tau & (\text{CT-LET}) \\
\Gamma; \varphi \vdash \lambda y^\tau.N : \tau & (\text{CT-SUB}) \\
\Gamma; \varphi \vdash N : \tau & (\text{CT-SUB}) \\
\end{align*}
\]

return value of the primitive function \(c\) applied to an argument \(v\) if \(v\) meets the constraint of the argument of \(c\), and otherwise undefined. We denote \(N \uparrow\) if there exists an infinite reduction sequence from \(N\).

The substitution for cast terms is defined in the standard manner, except that the implicitly-annotated type information and the predicate in the assertion need to be updated as well. As can be seen in the definition of the cast term reduction, these implicitly-annotated types are only required for the sake of formalization and ignored at runtime.

We also introduce the type derivation rules for the cast terms \(\Gamma; \varphi \vdash N : \tau\) in Figure 9. This relation is used in the discussion of the soundness of the type system later in Section 2.3. The quadruple relation \(\Gamma; \varphi \vdash N : \tau\) denotes that a cast term \(N\) has type \(\tau\) under a type environment \(\Gamma\) and a logical context \(\varphi\). The logical context \(\varphi\) holds the information of logically valid predicates at respective program points. New predicates are added at the then branch and the else branch of (CT-IF), and the post-assertion cast term in (CT-Ass). The subsumption is allowed in (CT-SUB) by the subtyping relation \(\Gamma; \varphi \vdash \tau_1 <: \tau_2\) (Figure 10), which is defined in a standard manner.
2.3 Typing Rules

Inserting Assertions Next, we discuss the typing rules for the source language and the assertion insertion into it. Figure 11 defines the type judgment and cast insertion relation. The intuition of 5-ary relation $\Gamma; \phi \vdash M \Rightarrow N : \tau$ is: under a type environment $\Gamma$ and a logical context $\phi$, a term $M$ translates to a cast term $N$ and has type $\tau$. If we ignore the part “$\Rightarrow N$” and replace the gradual subtyping relation `$\preceq$` with the standard subtyping relation on refinement types (Figure 10), our type system is a standard refinement type system. Thus, the main novelty in the rules in Figure 11 lies in the use of the consistent subtyping relation $\Gamma; \phi \vdash \tau_1 \preceq \tau_2 \Rightarrow N$, which is explained below.

The consistent subtyping relation $\Gamma; \phi \vdash \tau_1 \preceq \tau_2 \Rightarrow N$ (Figure 12) is used in the cast insertion relation to guarantee that there exists a value that has both of the types $\tau_1$ and $\tau_2$ under $\Gamma$ and $\phi$, and to produce an assertion term $N$ that checks at runtime if a value that is statically known to be of type $\tau_1$ can be used as a value of type $\tau_2$.

The rule for the base case (CAST-BASE) checks if there exists a value, and an assignment of the values to the variables in the type environment, that satisfies both $\tau_1$ and $\tau_2$. This intuitively holds if $\tau_1$ is castable to $\tau_2$ for some runtime values. The rule also produces a lambda function that implements the cast with an assertion. It is defined in such a way that $\varphi_2$ can always be used as the content of the assertion $\varphi'$, but $\text{true}$ can also be used for $\varphi'$ if $\varphi_1$ implies $\varphi_2$. Note that we cannot use $\varphi_2$ as the content of the assertion in the definition, or otherwise Proposition 1 does not hold.

The rule for the function types (CAST-FUN) recursively checks the castability of the argument types and the return types and combines the assertion terms for them. Notice how the subsumption for the return types $\tau_2$ and $\tau_4$ has the

---

5 This can be understood as the refinement-type version of the differential subtyping in [22], although in the implementation we do not calculate the “difference” between $\varphi_1$ and $\varphi_2$ for $\varphi'$ in the assertion unless $\varphi_1$ implies $\varphi_2$ (and thus $\varphi'$ can be $\text{true}$).
assigned a function type from the pre-assertion type to the post-assertion type. The soundness is based on the fact that if the source program is type-safe, all the assertions in the type system are valid. We conclude this section with a note on the soundness of our type system.

The consistent subtyping relation can be seen as a gradualization of the blame-subtyping theorem, one of the criteria for gradual type systems. In fact, when a type \( \tau_1 \) is a subtype of another type \( \tau_2 \), it is possible that the assertion term generated by casting \( \tau_1 \) to \( \tau_2 \) only contains assertions that always succeed, which can be erased by some optimization. The following proposition states this fact. Note that this corresponds to the blame-subtyping theorem, one of the criteria for gradual typing presented in [26].

**Proposition 1.** \( \Gamma; \varphi \vdash \tau_1 \leftarrow \tau_2 \) implies \( \Gamma; \varphi \vdash \tau_1 \preceq \tau_2 \rightarrow N \) for some \( N \) where all the assertions in \( N \) are of the form \( \text{assert}(\text{true}); N' \).

### Type Safety
We conclude this section with a note on the soundness of our type system. The soundness is based on the fact that if the source program is well-typed, the program after the assertion insertion is also well-typed.

The most critical part of the proof is to prove the assertion term can be assigned a function type from the pre-assertion type to the post-assertion type.

---

6 Although the meit of two argument types \( \tau_1 \cap \tau_3 \) in the type environment. The meet of two types (Figure 12) is defined as a conjunction of the refinement predicates.
\[
\{ x : B \mid \varphi_1 \} \cap \{ x : B \mid \varphi_2 \} = \{ x : B \mid \varphi_1 \land \varphi_2 \}
\]
\[
(x : \tau_1 \rightarrow \tau_2) \cap (x : \tau_3 \rightarrow \tau_4) = x : (\tau_1 \cap \tau_3) \rightarrow (\tau_2 \cap \tau_4)
\]

\[
\models \exists \text{BT}(\Gamma), x : B. \Phi(\Gamma) \land \varphi \land \varphi_1 \land \varphi_2 \models (\varphi' \Leftrightarrow \varphi_2)
\]

\[
\Gamma; \varphi \vdash \{ x : B \mid \varphi_1 \} \preceq \{ x : B \mid \varphi_2 \} \rightsquigarrow \lambda x : (v : B \mid \varphi_1). \text{assert}(\varphi'); x
\]

\text{(CAST-BASE)}

\[
\Gamma; \varphi \vdash \tau_3 \preceq \tau_1 \rightsquigarrow N_1 \quad \Gamma; x : \tau_1 \cap \tau_3; \varphi \vdash \tau_2 \preceq \tau_4 \rightsquigarrow N_2
\]

\text{(CAST-FUN)}

\[
\Gamma; \varphi \vdash x : \tau_1 \rightarrow \tau_2 \preceq x : \tau_3 \rightarrow \tau_4 \rightsquigarrow \lambda f : x : \tau_1 \rightarrow \tau_2, \lambda x : \tau_3. \text{let } y \text{ in } f y \text{ in } N_1 \text{ in } \text{let } z \text{ in } f z = f y \text{ in } N_2 z
\]

Fig. 12. Definition of the consistent subtyping relation \( \Gamma; \varphi \vdash \tau_1 \preceq \tau_2 \rightsquigarrow N \).

**Lemma 1.** \( \Gamma; \varphi \vdash \tau_1 \preceq \tau_2 \rightsquigarrow N \) implies \( \Gamma; \varphi \vdash x : \tau_1 \rightarrow \tau_2 \) for some variable \( x \) that does not occur in \( \tau_2 \).

This is an immediate consequence of Lemma 6 in the appendix. With Lemma 1, we can prove that the assertion-inserted program can be assigned the same type as that of the original program.

**Lemma 2 (Assertion Insertion Preserves Types).** \( \Gamma; \varphi \vdash M \rightsquigarrow N : \tau \) implies \( \Gamma; \varphi \vdash N : \tau \).

We can also prove the standard progress and preservation properties under a reasonable assumption that the types of the primitive functions are properly defined as follows (see the appendix for the proofs).

**Assumption 1** \( \vdash c v : \tau \) implies \( \text{ev}(c, v) \) is defined and \( \vdash \text{ev}(c, v) : \tau \)

Combining Lemma 2 with the progress and preservation properties, we obtain the type safety as follows.

**Theorem 1 (Type Safety).** With Assumption 1, \( \emptyset; \text{true} \vdash M \rightsquigarrow N : \tau \) implies \( N \rightsquigarrow^{*} v \) for some \( v \), \( N \not\downarrow \), or \( N \rightsquigarrow^{*} \text{error} \).

The type safety property states that a well-typed program does not cause untrapped dynamic errors. The only case where a cast-inserted program causes untrapped errors is when the result of an application of a primitive function is undefined (i.e., \( \text{ev}(c, v) \) is undefined). The type safety property ensures that such untrapped errors do not happen for well-typed terms as long as the \( \text{ty}(c) \) is defined appropriately.

## 3 Gradual Guarantee

In a standard gradual type system, programs are compared by their precision, or the amount of information contained in the type annotations. This notion is
used to define the gradual guarantee [26], which is the core property of gradual typing. The gradual guarantee comes in two parts. The first one is called static gradual guarantee, which states that decreasing the precision of type annotation from a well-typed program still preserves the typeability of the program at a less precise type. The second one is called dynamic gradual guarantee, which claims that a less precise program behaves the same as the more precise one with fewer assertion errors.

Below we first define the precision for the language introduced in Section 2. We then show that our type system satisfies the gradual guarantee.

**Precision.** Figure 13 defines the precision relation $\overline{x} \vdash \tau_1 \sqsubseteq \tau_2$ on types by using the logical implication between the refinement predicates. The sequence of variables $\overline{x}$ keeps the variables that may appear in the refinement predicates. For example, the following is an example of the type precision relation for the base type.

$\vdash \{ x : \text{tensor} | x.\text{shape} = [3] \} \sqsubseteq \{ x : \text{tensor} | \text{len}(x.\text{shape}) = 1 \}$

Figure 13. Precision relation of types and type environments.

Note that in the rule (Prec-Fun), the precision of the argument type and the return type are compared independently; the type information on $x$ is not used in the comparison of the return types. This is in contrast with the rule (Sub-Fun) in Figure 10 for subtyping. Figure 13 also extends the relation to $\Gamma \sqsubseteq \Gamma'$ on type environments. The precision relation is also extended to the relation $\overline{x} \vdash M \sqsubseteq M'$ on terms, by the rules in Figure 14. Here, $\overline{x}$ is the sequence of variables in scope. Finally, we define the precision relation of the cast terms in Figure 14. Unlike the term precision relation (Figure 14), the precision relation $\Gamma; \varphi \vdash N_1 \sqsubseteq N_2$ on cast terms requires the type environment $\Gamma$ and the logical context $\varphi$ in the judgement, and the refinement extraction from the type environment $\Phi(\Gamma)$ is used in the rule (PC-Assert). We also assume the following property on the evaluation of the primitive functions.

**Assumption 2** If $\text{ev}(c, v_2)$ and $\text{ev}(c, v_1)$ are both defined, then $v_1 \sqsubseteq v_2$ implies $\text{ev}(c, v_1) \sqsubseteq \text{ev}(c, v_2)$

Intuitively, the precision of cast terms are designed in such a way that, when $\emptyset; \text{true} \vdash N_1 \sqsubseteq N_2$ holds, the assertions in $N_1$ is more strict than that of $N_2$, and therefore the dynamic checks in $N_1$ is more likely to fail than in $N_2$. The following two propositions state this intuition (the proofs are found in the appendix).
\[\bar{x} \vdash M_1 \sqsubseteq M_2\]

\[
\begin{array}{ccc}
\bar{y} \vdash \tau_1 \sqsubseteq \tau_2 & \bar{y}, x \vdash M_1 \sqsubseteq M_2 & \bar{y} \vdash M_1 \sqsubseteq M_2 \\
\bar{y} \vdash \lambda x: \tau_1 . M_1 \sqsubseteq \lambda x: \tau_2 . M_2 & \bar{y} \vdash (M_1 : \tau_1) \sqsubseteq (M_2 : \tau_2)
\end{array}
\]

\[\text{PM-LAM}\]

\[\text{PM-ANNOT}\]

\[\Gamma; \varphi \vdash N_1 \sqsubseteq N_2\]

\[\forall \text{BT}(\Gamma). \Phi(\Gamma) \land \varphi \land \varphi_1 \Rightarrow \varphi_2 \quad \Gamma; \varphi \land \varphi_1 \vdash N_1 \sqsubseteq N_2 \quad \Gamma; \varphi \vdash \text{assert}(\varphi_1); N_1 \sqsubseteq \text{assert}(\varphi_2); N_2\]

\[\text{PC-ASSERT}\]

Fig. 14. Selected rules for the precision relation on terms and cast terms (the full definition is found in the appendix Figure 18 and Figure 19 respectively).

**Proposition 2.** Suppose \(\varnothing; \text{true} \vdash N_1 : \tau \) and \(\varnothing; \text{true} \vdash N_2 : \tau'.\) Then, \(\varnothing; \text{true} \vdash N_1 \sqsubseteq N_2 \) and \(N_1 \rightarrow N'_1 \) imply \(N_2 \rightarrow N'_2 \) and \(\varnothing; \text{true} \vdash N'_1 \sqsubseteq N'_2\) for some \(N'_2\).

**Proposition 3.** Suppose \(\varnothing; \text{true} \vdash N_1 : \tau \) and \(\varnothing; \text{true} \vdash N_2 : \tau'.\) Then, \(\varnothing; \text{true} \vdash N_1 \sqsubseteq N_2 \) and \(N_2 \rightarrow N'_2\) imply either of the following.

- \(N_1 \rightarrow N'_1\) and \(N'_1 \sqsubseteq N'_2\) for some \(N'_1\)
- \(N_1 \rightarrow \text{error}\)

**Gradual Guarantee.** We show that our system satisfies the gradual guarantee [26]. First, we prove that the consistent subtyping relation \(\Gamma; \varphi \vdash \tau_1 \sqsubseteq \tau_2 \) is upper-closed with respect to the precision relation \(\bar{x} \vdash \tau_1 \sqsubseteq \tau_3\) on types.

**Lemma 3.** \(\Gamma; \varphi \vdash \tau_1 \sqsubseteq \tau_2 \) \(\Rightarrow\) \(N_1\), \(\text{dom}(\Gamma) \vdash \tau_1 \sqsubseteq \tau_3\), \(\text{dom}(\Gamma) \vdash \tau_2 \sqsubseteq \tau_4\) and \(\Gamma \subseteq \Gamma'\) implies \(\Gamma'; \varphi \vdash \tau_3 \sqsubseteq \tau_4 \) \(\Rightarrow\) \(N_2\) for some \(N_2\).

We can further prove that the cast term \(N_2\) in the statement of Lemma 3 is less precise than the original cast term \(N_1\) as follows.

**Lemma 4.** Suppose \(\Gamma \subseteq \Gamma', \text{dom}(\Gamma) \vdash \tau_1 \subseteq \tau'_1\) and \(\text{dom}(\Gamma) \vdash \tau_2 \subseteq \tau'_2\). Then, \(\Gamma; \varphi \vdash \tau_1 \sqsubseteq \tau_2 \) \(\Rightarrow\) \(N\) and \(\Gamma; \varphi \vdash \tau'_1 \subseteq \tau'_2 \) \(\Rightarrow\) \(N'\) implies \(\Gamma; \varphi \vdash N \sqsubseteq N'\).

Using the above properties, we can prove the following lemma which constitutes the core part of the proof of the gradual guarantee.

**Lemma 5.** \(\Gamma \subseteq \Gamma', \text{dom}(\Gamma) \vdash M \subseteq M'\) and \(\Gamma; \varphi \vdash M \rightarrow N : \tau\) imply \(\Gamma'; \varphi \vdash M' \rightarrow N' : \tau'\), \(\Gamma; \varphi \vdash N \subseteq N'\) and \(\text{dom}(\Gamma) \vdash \tau \subseteq \tau'\) for some \(N'\) and \(\tau'\).

Finally, we can show the static and dynamic gradual guarantee as follows.

**Theorem 2 (Static gradual guarantee).** \(\varnothing \vdash M_1 \subseteq M_2\) and \(\vdash M_1 : \tau_1\) imply \(\vdash M_2 : \tau_2\) and \(\varnothing \vdash \tau_1 \sqsubseteq \tau_2\) for some \(\tau_2\).
Proof. This follows immediately from Lemma 5. □

Theorem 3 (Dynamic gradual guarantee). Suppose ⊢ M₁ ⊑ M₂ and ⊢ M₁⇝ N₁ : τ₁. Then, there exist N₂ and τ₂ that satisfy all of the following.

- ⊢ M₂⇝ N₂ : τ₂.
- N₁ ⊑ v₁ implies N₂ ⊑ v₂ for some v₂.
- N₁ ⊨ implies N₂ ⊨.
- N₂ ⊑ v₂ implies N₁ ⊑ v₁ and v₁ ⊑ v₂ for some v₁, or N₁ ⊨ error.
- N₂ ⊨ implies N₁ ⊨ or N₁ ⊨ error.

Proof. By Lemma 5, ⊢ M₂⇝ N₂ : τ₂ holds for some N₂ and τ₂ where ⊢ N₁ ⊑ N₂ and ⊢ τ₁ ⊑ τ₂. Also, from Lemma 2, we obtain ⊢ N₁ : τ₁ and ⊢ N₂ : τ₂. Using Proposition 2, N₁ ⊑ v₁ implies N₂ ⊑ v₂ for some v₂ such that v₁ ⊑ v₂. Also, N₁ ⊌ implies N₂ ⊌. Using Proposition 3, N₂ ⊑ v₂ implies N₁ ⊑ v₁ for some v₁ such that v₁ ⊑ v₂, or N₁ ⊨ error. Also, N₂ ⊌ implies N₁ ⊌ or N₁ ⊨ error. □

4 Best-Effort Type Inference

Thanks to our combination of gradual typing and hybrid checking described in the previous sections, a type inference procedure need not necessarily output the most precise types. It is allowed to perform type inference only in a best-effort manner, and the results in the previous sections do not depend on the particular design of the type inference procedure. Nevertheless, it is desirable for the procedure to infer reasonably good types. In this section, we report a specific design of the type inference procedure, which we have implemented in our prototype system GraTen; as reported in the Section 5, our procedure works reasonably well for actual deep learning programs.

4.1 Overview of Type Inference and Checking in GraTen

The type checking in GraTen consists of the following three phases: (1) simple type inference, (2) best-effort refinement type inference, and (3) consistent subtyping checking and assertion insertion.

In the first phase, GraTen performs the simple type inference using the standard Hindley-Milner algorithm and annotates the AST with the inferred simple types of each node.

In the second phase, GraTen first collects all the consistent subtyping constraints of the form Γ; ϕ ⊢ τ₁ ⊑ τ₂ ⊨ N from the source program. When it encounters AST nodes whose refinement type cannot be constructed directly, GraTen generates template refinement types using the simple types inferred in the previous phase. Template refinement types may contain variables for undetermined predicates (referred to as predicate variables).

Using the collected constraints, GraTen then tries to find a solution for all of the predicate variables with its hand-made constraint solver. The constraint
solving takes place on every let binding to allow let-polymorphism on shapes. We discuss the detail of the implementation of the solver in the next subsection, but at a high level, the solver tries to find such a solution that:

- only general types are inferred, as otherwise it could result in rejecting well-typed programs.
- $\Gamma; \varphi \vdash \tau_1 <: \tau_2$ holds for as many constraints $\Gamma; \varphi \vdash \tau_1 \preccurlyeq \tau_2 \rightsquigarrow N$ as possible. This is to make the cast term $N$ consist of trivial assertions (which can statically be discharged to avoid run-time overheads; recall Proposition 1).

Given that the subtyping constraints can be expressed in the form of constrained Horn clauses (CHC) and not all the subtyping constraints need to hold, the problem above is essentially a CHC solving problem with weak constraints and maximality [21] where the optimization objective of the problem is defined by pointwise logical comparison of the solutions.

The constraint solver of GraTen does not always find a solution for all predicate variables. In such cases, GraTen assigns true to the undetermined predicate variables; that way, they will at least not invalidate the consistent subtyping constraints.

Note that GraTen does not take into account the consistent subtyping relation $\Gamma; \varphi \vdash \tau_1 \preccurlyeq \tau_2 \rightsquigarrow N$ itself when trying to find a solution, as we expect that it would be rare for a consistent subtyping $\Gamma; \varphi \vdash \tau_1 \preccurlyeq \tau_2 \rightsquigarrow N$ to hold when the subtyping relation $\Gamma; \varphi \vdash \tau_1 <: \tau_2$ does not hold. GraTen therefore defers the check of consistent subtyping constraints to the next phase.

In the third phase, GraTen checks the validity of consistent subtyping constraints using the inference results for the predicate variables from the previous phase. GraTen first attempts to simplify and verify the constraints by a hand-made solver, but it falls back on using z3 [5] with timeouts if it does not work. Simultaneously, it also generates the assertion terms and inserts them into the source program.

4.2 Heuristics of Best-Effort Type Inference

To solve the subtyping constraints explained above, we have implemented a hand-made constraint solver. GraTen does not use off-the-shelf SMT or CHC solvers such as Z3 [5], since the refinement predicates in GraTen often use complicated predicates on integer lists, for which standard SMT/CHC solvers cannot find a solution in a reasonable time. Also, while GraTen should infer general types (so as not to reject well-typed programs), those generic solvers are not biased towards generality and return any (non-general) solution that satisfies the constraints. This subsection describes the heuristics used in GraTen for constraint solving.

The preparation for the inference is already started when GraTen generates the template refinement types during the constraint collection. For each predicate variable generated, GraTen attaches the set of program variables it depends on, which is calculated from the type environment. This is used in the constraint
solving later to avoid assigning irrelevant predicates to the predicate variables. We denote predicate variables as $p_{\tilde{z}}(\bar{y})$, where $\bar{x}$ denotes the set of program variables it depends on and $\bar{y}$ denotes the parameters of the predicate variable.

After collecting the constraints, GRATen decomposes the subtyping constraints to constrained Horn clauses of the form $\varphi_1 \land \varphi_2 \Rightarrow \varphi_3$ following the definition of the subtyping relation (Figure 10). The notation $\varphi$ denotes a set of predicates, logically interpreted as the conjunction of the predicates. The first, second, and third set of predicates in the clause respectively corresponds to the predicates from the context $\Phi(\Gamma) \land \varphi$, the refinement of the type on the left and the refinement of the type on the right $\varphi_2$. We intentionally distinguish between $\varphi_1$ and $\varphi_2$ on the left-hand side of the clauses in describing the constraint solving algorithm. For example, let us reconsider the program in Figure 2. The subtyping constraints collected from the if expression of the program would be as follows, where $p$, $q$, and $r$ are the predicate variables generated for the type of $s$, $x$ and the if expression respectively.

\[
\begin{align*}
\Gamma; (s = 1) & \vdash \{\nu;\text{tensor} \mid q_{s,v}(\nu)\} : \{\nu;\text{tensor} \mid r_{s,x,v}(\nu)\} \\
\Gamma; (s \neq 1) & \vdash \{\nu;\text{tensor} \mid q_{s,v}(\nu)\} : \{\nu;\text{tensor} \mid \text{len}(\nu.\text{shape}) = 1\} \\
\Gamma; (s \neq 1) & \vdash \text{tensor}(\text{nth}(0, x.\text{shape})/s) : \{\nu;\text{tensor} \mid r_{s,x,v}(\nu)\}
\end{align*}
\]

where $\Gamma := [s \mapsto \{\nu;\text{int} \mid p_{s}(\nu)\}, x \mapsto \{\nu;\text{tensor} \mid q_{s,v}(\nu)\}]$

These constraints are decomposed into the following clauses.

\[
\begin{align*}
\{p_{s}(s), q_{s,x}(x), s = 1\} \land \{q_{s,v}(\nu)\} & \Rightarrow r_{s,x,v}(\nu) \\
\{p_{s}(s), q_{s,x}(x), s \neq 1\} \land \{q_{s,v}(\nu)\} & \Rightarrow \text{len}(\nu.\text{shape}) = 1 \quad (1) \\
\{p_{s}(s), q_{s,x}(x), s \neq 1\} \land \{
u.\text{shape} = \text{nth}(0, x.\text{shape})/s\} & \Rightarrow r_{s,x,v}(\nu)
\end{align*}
\]

From the clauses obtained as above, GRATen tries to find a solution for the predicate variables using an algorithm presented in Algorithm 1.

The algorithm processes the constraints by first trying to find a solution for predicate variables that occur on the right-hand side of a clause $\varphi_1 \land \varphi_2 \Rightarrow \varphi_3$ (Line 6-10), and then on the left-hand side of a clause (Line 11-15), and repeats it until either all of the constraints are solved or the constraints cannot be processed any further (Line 4). In Line 8 and Line 13, the set of program variables $\bar{x}$ of a predicate variable $p_{\bar{x}}$ is used to assign the predicates to the predicate variables\(^7\).

During the iteration, the constraints need to be occasionally updated with the current solutions $\theta$ by applying the substitution $\theta$ to all the predicates in the constraints. After that, we also simplify the set of clauses (with simplify in Algorithm 1) by removing the predicates from the right-hand side of a clause that trivially follows from the left-hand side, and by removing clauses whose right-hand side is empty. For example, a clause $\{\} \land \{x = 1\} \Rightarrow \{\}$ is simplified to $\{\} \land \{x = 1\} \Rightarrow \{\}$, and then removed from the set of clauses.

To illustrate the behavior of Algorithm 1, consider applying it to the clauses (1). During the first iteration of the while loop (Line 4), the first for loop (Line

\(^7\) The set of program variables used in predicates is defined following the standard definition of free variables, except that the program variables used in a predicate variable $p_{\bar{x}}$ is defined as $\bar{x}$.
6) exits with an empty \( \theta \) as \( r \) appears on the right-hand side of multiple clauses and cannot be resolved here due to the check at Line 7. In the next for loop (Line 11), \( \theta \) is updated to:

\[
[q_{s,\nu}(\nu) \mapsto (\text{len}(\nu.\text{shape}) = 1 \land q'_{s,\nu}(\nu))]
\]

where \( q_{s,\nu}(\nu) \) is a fresh predicate variable, and the constraints \( c \) would be updated as follows.

\[
\{p(s), \text{len}(x.\text{shape}) = 1\} \land \{q'_{s,x}(x), s = 1\} \land \{q'_{s,\nu}(\nu)\}
\]

\[
\{p(s), \text{len}(x.\text{shape}) = 1, q'_{s,x}(x), s \neq 1\} \land \{\nu.\text{shape} = [\text{nth}(0, x.\text{shape})/s]\} \Rightarrow r_{s,x,\nu}(\nu)
\]

The while loop exits after the second iteration, as no new predicate variables can be added to \( \theta \) and \( c = c' \) holds. Thus, we only obtain (2) from Algorithm 1. After the inference, GRATen assigns true to the remaining predicate variables \( p, q' \) and \( r \).

**Algorithm 1** Algorithm for calculating the solutions \( \theta \) to predicate variables from constrained Horn clauses \( c \).

**Input:** constrained Horn clauses \( c \)

**Output:** the mapping from predicate variables to its solution (predicates) \( \theta \)

1: procedure solve\((c)\)
2: Let \( \theta \) be an empty substitution
3: \( c' \leftarrow c \)
4: while \( c \neq \emptyset \) and \( c \neq c' \) do
5: \( c' \leftarrow c \)
6: for every clause of the form \( \varphi_1 \land \varphi_2 \Rightarrow p\tilde{x}(\tilde{y}) \) in \( c \) do
7: if \( p\tilde{x}(\tilde{y}) \not\notin \varphi_1' \) for any other \( \varphi_1' \land \varphi_2' \Rightarrow \varphi_3' \) in \( c \) then
8: \( \theta \leftarrow [p\tilde{x}(\tilde{y}) \mapsto \bigwedge \varphi_2'] \circ \theta \)
\( \triangleright \circ \) is a composition of mappings.
9: \( c \leftarrow \text{simplify}(\theta c) \)
10: for \( \varphi_1 \land \varphi_2 \Rightarrow \varphi_3 \) in \( c \) do
11: let \( \varphi_1' \) be the maximal subset of \( \varphi_1 \) that only uses variables in \( \tilde{x} \)
12: \( \theta \leftarrow [\varphi_1' \mapsto \bigwedge \varphi_2'] \circ \theta \)
13: for every predicate variable \( p\tilde{x}(\tilde{y}) \) in \( \varphi_1' \land \varphi_2 \) do
14: \( \theta \leftarrow [q\tilde{x}(\tilde{y}) \mapsto (\bigwedge \varphi_2') \land q\tilde{x}(\tilde{y})] \circ \theta \)
15: \( c \leftarrow \text{simplify}(\theta c) \triangleright \) Also updates the remaining items iterated by L11.
16: return \( \theta \)

## 5 Experiment

This section reports on experiments to evaluate the effectiveness of our approach by running our tool GRATen for the example programs bundled in the OCaml-Torch library [4]. We have also checked how type annotations changed the inference results.
5.1 Methods

**Input and Output of GraTen** GraTen takes an OCaml program and performs type checking with its best-effort type inference. If the type checking is successful, it returns the inferred types of top-level variables defined in the program, and the source program with necessary assertions inserted. Otherwise, the type checking fails with an error message.

The assertions are inserted into the output program only when they are needed. Namely, assertions are inserted into the places where the consistent subtyping $\Gamma; \varphi \vdash \tau_1 \trianglelefteq \tau_2 \rightarrow N$ is used only when $\Gamma; \varphi \vdash \tau_1 :<: \tau_2$ doesn’t hold (see Proposition 1).

Besides the source program, GraTen also reads the types of the library functions (including those of OCaml-Torch) from manually prepared stub files. For example, the type of `tr` (matrix transpose function) is defined as follows.

```ocaml
val tr : x:{ v:tensor | len v.shape = 2 } -> tensor([nth 1 x.shape; nth 0 x.shape])
```

Note that describing the types of some higher-order OCaml-Torch functions requires the polymorphic extension, which we sketch in Appendix A. For example, the type of `Layer.forward` is defined as follows.

$$\forall b_1:bool, b_2:bool.
\{x:{x:tensor | b_1} \rightarrow \{y:tensor | b_2\}\} \rightarrow \{x:{x:tensor | b_1} \rightarrow \{y:tensor | b_2\}\}$$

GraTen handles such types by instantiating the quantified parameters ($b_1$ and $b_2$ in the above case) with fresh predicate variables.

**Test Cases** We applied GraTen to programs under examples/ directory of the repository of OCaml-Torch. The list of programs tested is shown in Table 1. Since some programs use features of OCaml or OCaml-Torch that are not yet supported by GraTen, they were modified not to use such features without changing the structure of the neural network. Major modifications added to the target programs are listed below. Other smaller syntactic modifications can be found in the supplementary materials.

(M1) Replacing or removing type-polymorphic functions. Some functions that create loops such as `List.foldl` are replaced with recursive functions. Others such as `no_grad` are replaced with the type-instantiated versions.

(M2) Removing use of non-integer lists, especially tensor lists and layer lists. As a result, two list-taking primitive functions are removed. One is `Tensor.cat`, which takes a list of tensors and returns the concatenation of them. It is replaced with a variant `Tensor.cat_`, which takes only two tensors. The other is `Layer.sequential`, which takes a list of layers and returns a layer that sequentially applies all the input layers.

8 [https://github.com/LaurentMazare/ocaml-torch/tree/a6499811f4/examples](https://github.com/LaurentMazare/ocaml-torch/tree/a6499811f4/examples)

9 Functions that take a tensor and return a tensor.
(M3) Replacing mutable float objects with 0-dimensional tensors, as GraTen does not support reference types.

As an example of (M1) and (M2), consider the following function, which creates a list of linear layers and returns a new layer that applies all the layers in the list.

```ml
let rec loop i xs =
  if i = 0 then Layer.id xs
  else loop (i-1) xs ~is_training |> Layer.linear vs ~input_dim:i (i+1)
in Layer.of_fn (loop num_layers)
```

The i-th layer in the list takes a tensor whose last dimension is size i+1, and returns a tensor of the same shape except that the last dimension is changed to i+2. By the modifications (M1) and (M2), the above function definition is replaced with:

```ml
let f vs ~num_layers =
  let rec loop i xs =
    if i = 0 then Layer.id xs
    else loop (i-1) xs ~is_training |> Layer.linear vs ~input_dim:i (i+1)
  in Layer.of_fn (loop num_layers)
```

Some programs in the examples/ directory are excluded from the test cases for the following reasons.
- `neural_transfer` uses a library function `Vgg.vgg16_layers` whose type cannot be described in GraTen; the relation between its inputs and its output tensor's shape could not be expressed in the syntax supported by GraTen.
- Programs `dqn.ml`, `dqn_atari.ml` and `dqn_pong.ml` in reinforcement-learning use queues which are not supported in GraTen yet.
- `env_gym_pyml.ml` and `venv_env_gym_pyml.ml` under reinforcement-learning use Python objects whose verification is not the scope of this paper.
- `reinforcement-learning/policy_gradient.ml` uses mutable lists which cannot be replaced with another datatype already supported in GraTen.
- `yolo/darknet.ml` and `translation/lang.ml` use hash tables which are not supported in GraTen yet.
- `translation/dataset.ml` and `translation/lang.ml` are irrelevant as tensor objects do not appear in them.

**Evaluation** We evaluated the best-effort inference of GraTen on the following three aspects.

First, we counted the assertions inserted into the original program when GraTen is used for the target program. Since the assertions indicate the program points that could fail at runtime, the user of GraTen would wish to pay attention to the location and the number of inserted assertions and try to decrease them.
Second, we counted the minimum number of type annotations required to type-check the program with minimum assertions inserted. This is for evaluating the realistic programmers’ burden of trying to statically verify the program with type annotations. The annotations were added in such a way that the types of the functions do not lose the original generality. The type annotations are counted by the number of refinement types with non-true refinement predicates in them. For example, the following annotation counts as 3 because the refinement of the input tensor and the two output tensors are not true, but the refinement of the annotation of the second argument bool is true.

\[
tensor([x]) \rightarrow \text{~is\_training:bool} \rightarrow tensor([x]) \ast tensor([x])
\]

Third, we also measured the time taken by GraTen to analyze the unannotated and annotated programs. The experiments were conducted on a Linux machine with 12-core Intel i5-11400 (2.60GHz) and GraTen is implemented in Haskell with GHC version 9.0.2.

5.2 Experimental Results

Table 1 summarizes the experimental results. We analyze those results by the following three aspects: assertions, type annotations and analysis time.

**Inserted Assertions** Out of the 26 programs tested, 10 programs required no type annotations to type-check without assertions, and other 7 programs type-checked without assertions after adding appropriate type annotations. For the remaining 9 programs such as gan/began.ml and gan/gan_stability.ml, we could not eliminate all assertions, although some of them were removed after adding type annotations. The remaining assertions were due to the imprecise type signatures of some library functions. For instance, Torch.Serialize.load is a function that loads a tensor from a file and its type signature is defined as follows.

\[
\text{val load : ~filename:string} \rightarrow \text{tensor}
\]

The return type of load is simply defined as tensor since it is impossible to assume any properties about its shape. As a result, an assertion was inserted to check if the loaded tensor satisfies the requirement to run the program without uncaught errors. Even adding type annotations to the loaded tensor does not remove the assertion.

Some other functions are given imprecise types due to GraTen’s immature support of polymorphic data types. For example, the type of Tensor.stack is defined as follows because GraTen does not effectively support non-integer lists yet. Refining the return types of such functions is left as future work.

\[
\text{val stack : ~dim:int} \rightarrow \text{list (tensor)} \rightarrow \text{tensor}
\]
| Location under examples/ | LOC | Unannotated | Annotated |
|--------------------------|-----|-------------|-----------|
| char_rnn/char_rnn.ml     | 98  | 1.647 1 | #assert 2 | 0.664 0 |
| cifar/cifar_train.ml     | 72  | 0.311 0 | - - - - |
| cifar/densenet.ml        | 116 | 2.603 6 | #assert 2 | 1.304 0 |
| cifar/fast_resnet.ml     | 64  | 0.293 0 | - - - - |
| cifar/preact_resnet.ml   | 85  | 2.535 8 | #assert 5 | 0.346 0 |
| cifar/resnet.ml          | 78  | 2.597 8 | #assert 4 | 0.396 0 |
| gan/began.ml             | 220 | 1.581 1 | - - - - |
| gan/gan_stability.ml     | 224 | 4.441 40| #assert 2 | 1.410 2 |
| gan/mnist_cgan.ml        | 117 | 0.498 1 | - - - - |
| gan/mnist_dcgan.ml       | 136 | 1.418 4 | #assert 2 | 0.500 0 |
| gan/mnist_gan.ml         | 83  | 0.308 0 | - - - - |
| gan/progressive_growing_gan.ml | 118 | 0.734 0 | - - - - |
| gan/relativistic_dcgan.ml| 171 | 0.650 1 | - - - - |
| jit/load_and_run.ml      | 16  | 0.214 1 | - - - - |
| min-gpt/miningpt.ml      | 207 | 3.036 8 | #assert 6 | 2.686 0 |
| mnist/conv.ml            | 53  | 0.250 0 | - - - - |
| mnist/linear.ml          | 50  | 0.235 0 | - - - - |
| mnist/nn.ml              | 39  | 0.210 0 | - - - - |
| pretrained/finetuning.ml | 69  | 0.294 0 | - - - - |
| pretrained/predict.ml    | 68  | 0.303 2 | - - - - |
| reinforcement-learning/a2c.ml | 105 | 0.418 0 | - - - - |
| reinforcement-learning/ppo.ml | 249 | 0.438 0 | - - - - |
| reinforcement-learning/rollout.ml | 91  | 0.734 9 | #assert 5 | 0.425 1 |
| translation/seq2seq.ml   | 258 | 3.800 11 | #assert 34 | 1.023 3 |
| vae/vae.ml               | 78  | 1.233 4 | #assert 10 | 0.312 0 |
| yolo/yolo.ml             | 144 | 1.027 4 | #assert 1 | 0.985 3 |

**Table 1.** Results of running GraTen to the test cases. The second column is the size of the program after the modification. The third and fourth columns are the results for unannotated programs. The third column is the duration of the type-checking and the fourth column is the number of assertions inserted. From the fifth to the seventh columns are for the annotated programs. The fifth column is the number of annotations added to the program.
Patterns of Added Type Annotations

As we added type annotations to the test cases, we observed that the program points that require type annotations have similarities. All of the type annotations fall into one of the following patterns.

(P1) Branches i.e., if expressions and match expressions with multiple branches (e.g., Figure 4 in Section 1).

(P2) Recursive functions. For example, loop in translation/seq2seq.ml is annotated as follows.

```ml
let rec loop
  : ~state:tensore([1; enc.hidden_size])
  -> ~prevs:list ({ v:tensor | prod v.shape = 1 })
  -> ~max_length:int -> list ({ v:tensor | prod v.shape = 1 })
  = fun ~state ~prevs ~max_length -> ...
```

(P3) Higher-order shape-polymorphic arguments. For example, sample in char_rnn.ml is annotated as follows.

```ml
let sample ~dataset ~lstm
  ~linear:(linear : x:{ v:tensor | last v.shape = hidden_size } -> tensor(init x.shape @ [dataset.labels]))
  ~device = ...
```

(P4) Definition of record types. The current implementation of GraTen expects that the definition of record types describes the refinement types of each field.

(P5) Imprecise type signatures of primitive functions, or user-defined functions of dependent modules. For example, translation/seq2seq.ml has the following type annotation since the return type of Tensor.stack is only inferred to be tensor due to its imprecise type signature.

```ml
let enc_outputs : tensor([1; nth 1 v.shape; enc.hidden_size]) =
  Tensor.stack enc_outputs ~dim:1
```

The statically inferred type of enc_outputs here is tensor([1; enc.hidden_size]) list, so we would not need this type annotation if the type signature of Tensor.stack is appropriately defined. Since it is not possible to statically verify the correctness of these types of annotations, assertions would still be inserted after adding these annotations.

The first three patterns indicate that GraTen’s current best-effort type inference does not effectively infer precise refinements for branches, recursive functions and higher-order shape-polymorphic arguments. The fourth pattern (P4) would be inevitable when using record types. It remains as future work to exempt users from having to add type annotations for (P5). With such improvements, we believe that it will become easier to find program points that require type annotations for better inference.
**Number of Type Annotations** There is no correlation between the number of assertions inserted into the unannotated program and the number of annotations needed to the program to minimize the number of assertions.

For example, adding two type annotations to `gan/gan_stability.ml` resulted in removing 38 assertions. This is because GRATen inferred an imprecise type for a helper function `resnet_block` without any type annotations, and it degraded the precision of the inference for the 24 callers of the function. Meanwhile, `translation/seq2seq.ml` required comparatively many type annotations as it has many definition of record types and several recursive functions with multiple inputs.

**Analysis Time** For all of the 11 annotated programs, GRATen’s type checking for annotated programs was faster than the unannotated counterparts. This would be because having more static information made it easier for GRATen to infer more precise types and resolve more subsumption constraints easily.

### 5.3 Discussions

In this subsection, we discuss the strengths, weaknesses and our perspective on the future development of our system.

**Performance of Best-Effort Inference** As reported in the previous subsection, the best-effort inference of GRATen does not infer precise types for branches, recursions and higher-order shape-polymorphic arguments. While this may seem unsatisfying at a glance, the aim of this research is not to develop a perfect inference algorithm, but to propose a method that can work on unannotated programs and allows users to work interactively with the type checker to gradually add type annotations. With this respect, we believe that GRATen has achieved desirable results since it will be easy for the user to find out where to add type annotations. This is because (1) the inserted assertions can inform the user of the location of potential dynamic errors, and (2) all of the required type annotations would fall into one of the patterns listed in the previous section and thus should be predictable.

**Lists of Tensors and Layers** As of now, the refinement inference for lists in GRATen is limited to integer lists. Meanwhile, lists of tensors or lists of functions are commonly used in deep learning programs: `Tensor.cat` and `Tensor.stack` both take a list of tensors and return their concatenation, and `Layer.sequential` takes a list of layers (functions that take and return a tensor) and returns their composition.

A potential approach to support these library functions would be to add new refinement predicates for tensors lists or layer lists. For example, we can add a predicate `composable(x, S_1, S_2)` which means that the composition of a list of layers `x` takes a tensor of shape `S_1` and returns a tensor of shape `S_2`. The type of
Layer.sequential would be expressed with the shape polymorphic extension (see Appendix A) as follows.

\[
\text{val sequential : forall S1 S2.}
\{ v:list(tensor -> tensor) | composable(x,S1,S2) \}
\rightarrow tensor(S1) \rightarrow tensor(S2)
\]

To practically infer composable predicate for layer lists, we would need to change the type-instantiated versions of list-manipulating functions as well. For instance, the type of the cons function for layers would need to be defined as follows.

\[
\text{val cons_layers}
: \text{forall S1 S2 S3. (tensor(S1) -> tensor(S2))}
\rightarrow \{ v:list(tensor -> tensor) | composable(v, S2, S3) \}
\rightarrow \{ v:list(tensor -> tensor) | composable(v, S1, S3) \}
\]

**Reporting Incorrect Type Annotations** Since our type system sees the standard refinement types as gradual, some users might find the behavior of GraTen unexpected in some cases. Consider the following function f which takes a matrix and returns a matrix obtained by transposing the input. Suppose that the programmer mistakenly annotated the return value of f to have the same shape as the input matrix.

\[
\text{let f x = (tr x : tensor(x.shape))}
\]

Although this type annotation does not hold in general, this program is not rejected by our type system because the annotation can hold if the input x is a square matrix. GraTen would output the following program with an assertion.

\[
\text{let f x = (fun y -> assert(y.shape = x.shape); y) (tr x)}
\]

To avoid such a situation, it would be possible to extend the type system with types with fully statically known refinements, and let the annotated types be interpreted as such.

### 6 Related Work

**Tensor Shape Checking in Deep Learning Programs.** The problem of tensor shape checking has been studied for decades by various contexts such as the numeric analysis [7,2] and the array-oriented languages with rank polymorphism [28,27,12]. Tensor shape checking for deep learning programs is still a new challenge because the shapes can be more complicated, and a variety of methods have been proposed both in academia and in industry.

Some tools statically check tensor shapes with advanced type systems. Hasktorch [3] is a Haskell binding of libtorch [19] which provides a mode that statically checks tensor shapes. Since they use the type-level programming feature of Haskell to implement the tensor shapes, tensor shapes are not first-class objects.
As a result, programs such as the one in Figure 1 cannot be expressed since it is impossible to define the function $f$ whose type depends on the first-class object $s$. Relay [24,23] is an IR for deep learning compilers with a rich type system for tensor shape with type inference. Both Relay and Hasktorch support dynamic shape as a wild card in the static shape checking.

Apart from the type-based verification methods, some tensor shape error detection tools also take a static approach. Pythia [16,6] statically detects shape fault for TensorFlow [1] programs by keeping track of the tensor shapes throughout the program using value-flow analysis. The tracking of shape is in a best-effort manner, allowing the shape inference results to be “unknown” in some cases. The analysis crucially relies on the programming practice in TensorFlow to annotate tensor shapes as much as possible.

Other static checking tools took an approach that uses symbolic execution to collect constraints from the program and verifies it with a solver; Tensors Fitting Perfectly [20] and PyTea [14] are on this approach. Both methods remove loops from the program in an ad-hoc manner based on a reasonable assumption for the program.

Lastly, some took dynamic approaches to provide lightweight shape fault detection. ShapeFlow [30] is an abstract interpreter of TensorFlow programs; it shares the same APIs as TensorFlow but only calculates the shape of tensors. Users can run the analysis by replacing the import of TensorFlow with ShapeFlow in the target program, which executes more efficiently than the original TensorFlow program. Elichika [13] uses a similar method to ShapeFlow with a feature to display the interpreted shapes with a symbolic expression. These dynamic approaches enable quick analysis and require no type annotations, but provide no guarantee for untested inputs.

**Static and Dynamic Checking for Refinement Types.** Earlier work on dependent type system focused on decidable type checking and inference with restricted refinement logic [10,33,32,25]. Dynamic checking with contracts [18,9] offers expressive verification that cannot be covered with a static type system, but at a cost of runtime overhead. Naturally, the combination of static and dynamic checking has been actively explored by the successors of both parties.

Hybrid type checking [15], which our work is based on, extends the purely-dynamic method of using contracts by verifying specifications statically as much as possible. This method differs from ours in that it inserts a dynamic check only when the subtyping constraint is not proven to be valid or invalid. As a result, this method statically rejects the incorrectly annotated program that we discussed in Subsection 5.3, while our method accepts it with a dynamic check in the hope that a more precise type annotation will remove the need for a dynamic check. Our method can be understood as a variant of hybrid type checking with a focus on being gradual in adding type annotations.

The application of gradual typing to dependent type systems has also been studied [17,8]. Especially, gradual refinement types [17] is very similar to our type system in that it gradualizes only the predicate part of a refinement type system.
and the underlying simple type is static. One of the differences is that their system distinguishes statically-unknown refinement predicates with statically-known ones, while our system assumes that any refinement predicates can have a statically-unknown portion. For example, consider the following program:

\[
\text{let } f \ x (y : \{ \nu : \text{int} \mid \text{true} \}) = x/y
\]

This program is rejected in their system because the type annotation of \( y \) indicates that the programmer is confident that \( y \) can be any integers including 0; otherwise, the type annotation should have been \( \{ \nu : \text{int} \mid \ast \} \). Meanwhile, our system interprets the type annotation as not precise enough and accepts the program by inserting a dynamic check to \( y \). Intuitively, \( \{ x : B \mid \varphi \} \) in our type system translates to \( \{ x : B \mid \varphi \land \ast \} \) in gradual refinement types [17].

The type inference for gradual refinement types has been studied by Vazou et al. [29]. Their work restricts the refinement to liquid predicates [25] to maintain the decidability, while our work does not impose such a limitation.

7 Conclusion and Future Work

We presented an extension to the standard refinement type system which can be viewed as a gradual type system. The essence of this extension is the introduction of the consistent subtyping relation, which inserts to the source program assertions that checks statically-unverified properties at runtime. We also presented that the extended type system satisfies the refined criteria of gradual typing.

We then applied this type system for verifying tensor shapes with best-effort type inference. This application makes use of the property of the proposed type system that allows us to cover the limitation of the static best-effort analysis with dynamic checks. We also implemented a prototype type checker GRATEn and applied it with some of the example programs publicly available in OCaml-Torch repository. We observed that, thanks to the best-effort type inference, users would not be required too many type annotations to statically type-check the whole program, and it would not be difficult to find where to add type annotations to improve the inference.

We conclude with some ideas for future work.

– Extension with type polymorphism. As we observed in the experiments, type polymorphic functions are frequently used in realistic programs. Extending our type system with ML-style type polymorphism would make the type checker more practical.

– Application for imperative languages with a dynamic type system, like Python. In this paper, we have chosen OCaml as the target of the prototype to ensure that the input program is statically-typed. Python would, however, be a more attractive target since it is widely used in the machine learning community.

Acknowledgments We would like to thank anonymous referees for useful comments. This work was supported by JSPS KAKENHI Grant Number JP20H05703.
References

1. Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G.S., Davis, A., Dean, J., Devin, M., et al.: Tensorflow: Large-scale machine learning on heterogeneous distributed systems. arXiv preprint arXiv:1603.04467 (2016)
2. Abe, A., Sumii, E.: A simple and practical linear algebra library interface with static size checking. arXiv preprint arXiv:1512.01898 (2015)
3. contributors, H.: Hasktorch. http://hasktorch.org/ (2020), [Online; accessed 15-July-2021]
4. contributors, O.T.: Ocaml-torch. https://github.com/LaurentMazare/ocaml-torch (2020), [Online; accessed 05-July-2021]
5. De Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: International conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 337–340. Springer (2008)
6. Dolby, J., Shinnar, A., Allain, A., Reinen, J.: Ariadne: analysis for machine learning programs. In: Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages. pp. 1–10 (2018)
7. Eaton, F.: Statically typed linear algebra in haskell. In: Proceedings of the 2006 ACM SIGPLAN workshop on Haskell. pp. 120–121 (2006)
8. Eremondi, J., Tanter, É., Garcia, R.: Approximate normalization for gradual dependent types. Proceedings of the ACM on Programming Languages 3(ICFP), 1–30 (2019)
9. Findler, R.B., Felleisen, M.: Contracts for higher-order functions. In: Proceedings of the seventh ACM SIGPLAN international conference on Functional programming. pp. 48–59 (2002)
10. Freeman, T., Pfenning, F.: Refinement types for ML. In: Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation. pp. 268–277 (1991)
11. Garcia, R., Clark, A.M., Tanter, É.: Abstracting gradual typing. In: Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 429–442 (2016)
12. Gibbons, J.: Aplicative programming with naperian functors. In: Proceedings of the 1st International Workshop on Type-Driven Development. pp. 13–14 (2016)
13. Hattori, M., Sawada, S., Hamaji, S., Sakai, M., Shimizu, S.: Semi-static type, shape, and symbolic shape inference for dynamic computation graphs. In: Proceedings of the 4th ACM SIGPLAN International Workshop on Machine Learning and Programming Languages. pp. 11–19 (2020)
14. Jhoo, H.Y., Kim, S., Song, W., Park, K., Lee, D., Yi, K.: A static analyzer for detecting tensor shape errors in deep neural network training code. arXiv preprint arXiv:2112.09037 (2021)
15. Knowles, K., Flanagan, C.: Hybrid type checking. ACM Trans. Program. Lang. Syst. 32(2), 6:1–6:34 (2010). https://doi.org/10.1145/1667048.1667051, https://doi.org/10.1145/1667048.1667051
16. Lagouvardos, S., Dolby, J., Grech, N., Antoniadis, A., Smaragdakis, Y.: Static analysis of shape in tensorflow programs. In: 34th European Conference on Object-Oriented Programming (ECOOP 2020). Schloss Dagstuhl-Leibniz-Zentrum für Informatik (2020)
17. Lehmann, N., Tanter, É.: Gradual refinement types. In: Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages. pp. 775–788 (2017)
Using the refinement types, we can express the types of most of the shape polymorphic tensor functions. For example, ReLU is a function that takes a tensor and returns a tensor of the same shape as the input. In a type
system with shape polymorphism, the type of such function is expressed as 
\( \forall S : \text{int list}. \, \text{tensor}(S) \rightarrow \text{tensor}(S) \), whereas in our system, it is expressed as 
\( x : \text{tensor} \rightarrow \text{tensor}(x.\text{shape}) \).

There is, however, shape polymorphism that cannot be expressed in this 

style. Consider the type:

\[ \forall S_1, S_2 : (\text{tensor}(S_1) \rightarrow \text{tensor}(S_2)) \rightarrow \text{tensor}(S_1) \rightarrow \text{tensor}(S_2) \]

The application function \( \lambda f. \lambda x. f x \), for example, has this type. In our system 
presented so far, there is no way to present the equivalent of this type.

We therefore discuss an extension of our type system with explicit poly-
morphism on sizes and shapes. We introduce type schemes and redefined type 
evironments to map from variables to type schemes.

\[
T \text{ (type scheme)} := \forall x_1 : B_1, \ldots, x_n : B_n. \tau \\
\Gamma \text{ (type environment)} := \emptyset \mid \Gamma[x \mapsto T]
\]

We define the new typing relation as a transformation relation 
\( \Gamma; \Delta : \varphi \vdash M : \tau \Rightarrow M' \) which associates a polymorphically-typed program \( M \) with a 

monomorphically-typed program \( M' \):

\[
\Gamma(f) = \forall x_1 : B_1, \ldots, x_n : B_n. \tau \quad \text{BT}(\Gamma), \Delta \vdash t_i : B_i \text{ for each } i \]

\( \Gamma; \Delta : \varphi \vdash f : [t_1/x_1, \ldots, t_n/x_n] \tau \Rightarrow f \, t_1 \cdots t_n \) \hspace{1cm} (PT-VAR)

\( \Gamma; \Delta, x_1 : B_1, \ldots, x_n : B_n. ; \varphi \vdash M_1 : \tau_1 \Rightarrow M'_1 \) \hspace{1cm} \( x_1, \ldots, x_n \) do not occur in \( p \)

\( \Gamma(f ; \forall x_1 : B_1, \ldots, x_n : B_n. \tau_1) ; \Delta; \varphi \vdash M_2 : \tau_2 \Rightarrow M'_2 \) \hspace{1cm} \( \text{(PT-LET)} \)

Here, \( \Delta \) is a type environment for polymorphic shape variables.

Thus, a type scheme \( \forall x_1 : B_1, \ldots, x_n : B_n. \tau \) is mapped to a monomorphic (re-
finement) type \( x_1 : B_1 \rightarrow \cdots \rightarrow x_n : B_n \rightarrow \tau \). The term obtained by type inference is 
thus just a monomorphically-typed term, to which the theory of gradual tensor 
types developed in the previous subsections apply.

There is, however, a subtle conflict in the best-effort inference of type schemes 
and the coercion mechanism of gradual typing. In the rule \( \text{(PT-LET)} \), we need to 
infer appropriate \( t_1, \ldots, t_n \) and choosing wrong \( t_1, \ldots, t_n \) may cause cast failures, 
even if the original program is safe. For example, consider the following program:

\[
\text{let } g = \ldots \text{ (* complex function on tensors *) in } \ldots \text{ app } g \ldots
\]

Then, the function \text{app} can be assigned the following type scheme.

\( \forall S_1, S_2 : \text{int list}. (\text{tensor}(S_1) \rightarrow \text{tensor}(S_2)) \rightarrow \text{tensor}(S_1) \rightarrow \text{tensor}(S_2) \)

Type inference then tries to convert the caller \text{app} \, \text{g} \, x \ to a term of the form 
\text{app} \, t_1 \, t_2 \, \text{g} \, x. However, if \text{g} \ is a complex function for which the precise inference 
of shapes is difficult, then we will fail to infer appropriate parameters \( t_1, t_2 \). In
that case, we need to fall back to the safe side, and assign to \texttt{app} a less precise monomorphic type:

$$(\text{tensor} \rightarrow \text{tensor}) \rightarrow \text{tensor} \rightarrow \text{tensor}.$$ The caller \texttt{app g} is then just transformed to \texttt{app g x}.

The solution above (of falling back to monomorphic typing when type inference for callers fail) still has the following problems.

1. In the example above, other callers of \texttt{app} will also be monomorphically typed, which leads to imprecise type inference. For example, suppose \texttt{h} is statically known to have type \texttt{tensor([2; 3]) \rightarrow tensor([1])}. Based on the type scheme, we could infer the type of \texttt{app h} to be \texttt{tensor([2; 3]) \rightarrow tensor([1])}, but due to the presence of the caller \texttt{app g}, the imprecise type \texttt{tensor \rightarrow tensor} is inferred also for \texttt{app h}.

2. It is against the principle of modular type inference that whether a polymorphic type is assigned to \texttt{app} depends on callers.

A remedy to the problems above is to prepare both polymorphic and monomorphic versions for each polymorphic function. In the example above, thus \texttt{app} is transformed to two functions:

\begin{verbatim}
let app_poly s1 s2 f x = ...
let app_mono f x = ...
\end{verbatim}

where:

- \texttt{app_poly : \forall S_1, S_2. (tensor(S_1) \rightarrow tensor(S_2)) \rightarrow tensor(S_1) \rightarrow tensor(S_2)}
- \texttt{app_mono : (tensor \rightarrow tensor) \rightarrow tensor \rightarrow tensor}

Then we can transform \texttt{app g} and \texttt{app h} to \texttt{app_mono g} and \texttt{app_poly [2; 3] [1] h}.

\textbf{Remark 1.} The polymorphic extension we have discussed here may still not be suitable for the inference of some programs. For example, let \texttt{f} be a function of the following type:

\begin{verbatim}
f : \{ x: tensor | prod x.shape = 128 \} \rightarrow tensor
\end{verbatim}

and consider applying the above-mentioned \texttt{app} to \texttt{f}. Since it is impossible to infer the shape of the argument or the return value of \texttt{f}, the type of \texttt{app} falls back to that of \texttt{app_mono}. The type of \texttt{app f} is thus inferred to be \texttt{tensor \rightarrow tensor}, although we know that \texttt{app f} can be assigned the same type as \texttt{f}.

Function \texttt{Layer.forward} of OCaml-Torch is one of the most frequently used primitive function, and it has similar types as \texttt{app}. We noticed that not being able to infer the same type as \texttt{f} for \texttt{app f} in the above case critically degrades the precision of inference. Therefore, in the prototype implementation, we specially assign the following refinement-polymorphic type to \texttt{Layer.forward}:

\begin{verbatim}
\forall b_1: bool, b_2: bool.
(x: {x: tensor | b_1} \rightarrow {y: tensor | b_2}) \rightarrow x: {x: tensor | b_1} \rightarrow {y: tensor | b_2}
\end{verbatim}

and let \texttt{b_1} and \texttt{b_2} be instantiated with predicate variables. We leave the formal justification of such refinement-polymorphic types as future work.
B Complete Definitions and Proofs

B.1 Well-Formedness of Types

\[ \Delta \vdash_{\text{wf}} \varphi : \text{bool} \]
\[ \Delta \vdash_{\text{wf}} \{ x : B \mid \varphi \} \]
\[ \Delta, \text{BT}(x : \tau_1) \vdash_{\text{wf}} \tau_2 \]
\[ \Delta \vdash_{\text{wf}} x : \tau_1 \rightarrow \tau_2 \]

\( \Gamma \vdash_{\text{BT}} t : B \)
\[ \Delta \vdash_{\text{BT}} x : \Delta(x) \quad \Delta \vdash_{\text{BT}} \text{true} : \text{bool} \quad \Delta \vdash_{\text{BT}} \text{false} : \text{bool} \quad \Delta \vdash_{\text{BT}} n : \text{int} \]
\[ \Delta \vdash_{\text{BT}} s_i : \text{int} \quad (i = 1, 2) \quad \Delta \vdash_{\text{BT}} S_i : \text{int list} \quad (i = 1, 2) \quad \Delta \vdash_{\text{BT}} \varphi : \text{bool} \]
\[ \Delta \vdash_{\text{BT}} \varphi_1 \land \varphi_2 : \text{bool} \quad \Delta \vdash_{\text{BT}} \neg \varphi : \text{bool} \]
\[ \Delta \vdash_{\text{BT}} S_i : \text{int list} \quad (i = 1, 2) \quad \Delta \vdash_{\text{BT}} S_i : \text{int list} \quad (i = 1, 2) \quad \Delta \vdash_{\text{BT}} \text{broadcastable}(S_1, S_2) : \text{bool} \quad \Delta \vdash_{\text{BT}} \text{reshapeable}(S_1, S_2) : \text{bool} \]
\[ \Delta \vdash_{\text{BT}} \text{len}(S) : \text{int} \quad \Delta \vdash_{\text{BT}} s_i : \text{int} \quad (i = 1, 2) \quad \Delta \vdash_{\text{BT}} s_i : \text{int} \quad (i = 1, 2) \quad \Delta \vdash_{\text{BT}} \text{nth}(s, S) : \text{int} \quad \Delta \vdash_{\text{BT}} s : \text{int} \]
\[ \Delta \vdash_{\text{BT}} S_i : \text{int} \quad (i = 1, 2) \quad \Delta \vdash_{\text{BT}} s_i : \text{int} \quad (i = 1, 2) \quad \Delta \vdash_{\text{BT}} s_i : \text{int} \quad (i = 1, 2) \]
\[ \Delta \vdash_{\text{BT}} s_i + s_2 : \text{int} \quad \Delta \vdash_{\text{BT}} s_1 \times s_2 : \text{int} \quad \Delta \vdash_{\text{BT}} \frac{s_1}{s_2} : \text{int} \]

Fig. 15. Complete rules for the typing rules of predicates, shapes and sizes.

B.2 Semantics of Cast Terms

Figure 16 defines the full definition of the reduction of the cast terms \( N_1 \rightarrow N_2 \).

B.3 Properties about Type System

Proposition 4. (Proposition 1 in paper) \( \Gamma ; \varphi \vdash \tau_1 <: \tau_2 \) implies \( \Gamma ; \varphi \vdash \tau_1 \preceq \tau_2 \preceq N \) for some \( N \) where all the assertions in \( N \) are of the form \( \text{assert}(\text{true}); N' \).

Proof. By induction on \( \Gamma ; \varphi \vdash \tau_1 <: \tau_2 \).
Gradual Tensor Shape Checking (full version) 31

\[ [v/x]c = c \]
\[ [v/x]y = \begin{cases} v & (x = y) \\ y & (x \neq y) \end{cases} \]
\[ [v/x](N v') = ([v/x]N) ([v/x]v') \]
\[ [v/x](\lambda y^r. N) = \lambda y^{[v/x]r}. [v/x]N \]
\[ [v/x](\text{fix}(f^r, y, N)) = \text{fix}(f^{[v/x]r}, y, [v/x]N) \]
\[ [v/x](\text{assert}(x); N) = \text{assert}([v/x]x); [v/x]N) \]
\[ [v/x](\text{if } v_1 \text{ then } N_1 \text{ else } N_2) = \text{if } [v/x]v_1 \text{ then } [v/x]N_1 \text{ else } [v/x]N_2 \]
\[ [v/x](\text{let } y^r = N_1 \text{ in } N_2) = (\text{let } y^{[v/x]r} = [v/x]N_1 \text{ in } [v/x]N_2) \]

(We assume variables are appropriately alpha-renamed so that variables at different scopes do not collide)

\[ N_1 \rightarrow N_2 \]

\[ \text{assert}(true); N \rightarrow N \]
\[ \text{assert}(false); N \rightarrow \text{error} \]
\[ (\lambda x^r. N_1) v \rightarrow [v/x]N_1 \]
\[ \text{fix}(f^r, x, N_1)) v \rightarrow [v/x, \text{fix}(f^r, x, N_1)/f]N_1 \]
\[ c v \rightarrow \text{ev}(c, v) \]
\[ \text{let } x^r = v \text{ in } N \rightarrow [v/x]N \]
\[ \text{if } true \text{ then } N_1 \text{ else } N_2 \rightarrow N_1 \]
\[ \text{if } false \text{ then } N_1 \text{ else } N_2 \rightarrow N_2 \]

\[ C[N_1] \rightarrow \begin{cases} C[N_2] & (N_1 \rightarrow N_2) \\ \text{error} & (N_1 \rightarrow \text{error}) \end{cases} \]

\( C(\text{context}) ::= \square | C.N | v.C | \text{let } x = C \text{ in } N \)

Fig. 16. Substitution and reduction of the target language (full version of Figure 8).

- Case (Sub-Base).

\[ \vdash \forall BT(\Gamma), x:B. \Phi(\Gamma) \land \varphi \land \varphi_1 \Rightarrow \varphi_2 \]
\[ \Gamma; \varphi \vdash \{ x : B \mid \varphi_1 \} \triangleleft \{ x : B \mid \varphi_2 \} \]

Since \( \vdash \exists BT(\Gamma), x:B. \Phi(\Gamma) \land \varphi \land \varphi_1 \land \varphi_2 \) holds, the following is obtained as expected.

\[ \Gamma; \varphi \vdash \{ x : B \mid \varphi_1 \} \triangleleft \{ x : B \mid \varphi_2 \} \Rightarrow \lambda x^{(x:B[\varphi_1]).\text{assert}(true)}; x \]

- Case (Sub-Fun).

\[ \Gamma; \varphi \vdash \tau_5 \triangleleft \tau_3 \quad \Gamma; x : \tau_5; \varphi \vdash \tau_4 \triangleleft \tau_6 \]
\[ \Gamma; \varphi \vdash x : \tau_3 \rightarrow \tau_4 \triangleleft x : \tau_5 \rightarrow \tau_6 \]
From the induction hypothesis, there exists \( N_1 \) and \( N_2 \) such that \( \Gamma; \varphi \vdash \tau_5 \subseteq \tau_3 \prec N_1 \) and \( \Gamma, x : \tau_5; \varphi \vdash \tau_4 \subseteq \tau_6 \prec N_2 \) hold, and all the assertions in \( N_1 \) and \( N_2 \) are of the form \( \text{assert} (\text{true}) ; N' \). Since \( \Gamma, x : \tau_3 \cap \tau_5 \subseteq \Gamma, x : \tau_5 \) holds, \( \Gamma, x : \tau_3 \cap \tau_5 ; \varphi \vdash \tau_4 \subseteq \tau_6 \prec N_2 \) follows. Therefore, \( \Gamma; \varphi \vdash x : \tau_3 \to \tau_4 \subseteq x : \tau_5 \to \tau_6 \prec N \) holds for an \( N \) such that \( N \equiv \lambda \mathcal{f}_x : \tau_3 \to \tau_4 . \lambda \mathcal{x} : \tau_6 . \text{let } x \mathcal{f}_x \mathcal{g}_x = \mathcal{N}_1 x \text{ in let } \mathcal{y}^x = \mathcal{f} x \text{ in } \mathcal{N}_2 y \).

### B.4 Type Safety

**Lemma 6.** Let \( \text{Base}(\Gamma) \) and \( \text{self}(\tau, x) \) be defined as follows.

\[
\begin{align*}
\text{Base}(\emptyset) &= \emptyset \\
\text{Base}(\Gamma, x : \{ y : B \ | \ \varphi \}) &= \text{Base}(\Gamma), x : \{ y : B \ | \ \varphi \} \\
\text{Base}(\Gamma, x : y : \tau_1 \to \tau_2) &= \text{Base}(\Gamma) \\
\text{self}(\{ x : B \ | \ \varphi \}, y) &= \{ x : B \ | \ \varphi \land x = y \} \\
\text{self}(x : \tau_1 \to \tau_2, y) &= x : \tau_1 \to \tau_2
\end{align*}
\]

Then, \( \Gamma; \varphi \vdash \tau_1 \subseteq \tau_2 \prec N \) implies \( \text{Base}(\Gamma) ; \varphi \vdash N : \tau_1 \to \text{self}(\tau_2, x) \) for a variable \( x \) does not occur in \( \tau_2 \).

**Proof.** By induction on the derivation of \( \Gamma; \varphi \vdash \tau_1 \subseteq \tau_2 \prec N. \)

- **Case (Cast-Base).** The derivation must be of the following form (note \( \tau_1 \equiv \{ x : B \ | \ \varphi_1 \} \) and \( \tau_2 \equiv \{ x : B \ | \ \varphi_2 \} \)).

\[
\begin{align*}
\vdash & \exists \mathcal{B} \text{ (Base)} (\Gamma), x : B. \Phi(\Gamma) \land \varphi \land \varphi_1 \land \varphi_2 \\
\vdash & \forall \mathcal{B} \text{ (Base)} (\Gamma), x : B. \Phi(\Gamma) \land \varphi \land \varphi_1 \Rightarrow (\varphi' \equiv \varphi_2) \\
\Gamma; \varphi \vdash & \{ x : B \ | \ \varphi_1 \} \subseteq \{ x : B \ | \ \varphi_2 \} \prec \lambda x \{ x : B | \varphi_1 \} . \text{assert}(\varphi'); x \\
\end{align*}
\]

Noting \( \mathcal{B} (\Gamma) = \mathcal{B} (\text{Base}(\Gamma)) \) and \( \Phi(\Gamma) = \Phi(\text{Base}(\Gamma)) \), the following holds.

\[
\vdash \forall \mathcal{B} \text{ (Base)} (\Gamma), x : B. \Phi (\text{Base}(\Gamma)) \land \varphi \land \varphi_1 \land \varphi' \land y = x \Rightarrow [y/x] \varphi_2
\]

Therefore, the following proof tree concludes the proof.

\[
\begin{align*}
\text{Base}(\Gamma), x : \tau_1 ; \varphi \land \varphi' \vdash x : \{ y : B \ | \ y = x \} \\
\text{Base}(\Gamma), x : \tau_1 ; \varphi \land \varphi' \vdash \{ y : B \ | \ y = x \} < : \{ y : B | [y/x] \varphi_2 \land y = x \} \\
\text{Base}(\Gamma), x : \tau_1 ; \varphi \land \text{assert}(\varphi') \vdash x : \{ y : B | [y/x] \varphi_2 \land y = x \}
\end{align*}
\]

- **Case (Cast-Fun).** The derivation must be of the following form (note \( \tau_1 \equiv x : \tau_3 \to \tau_4 \) and \( \tau_2 \equiv x : \tau_5 \to \tau_6 \)).

\[
\begin{align*}
\Gamma; \varphi \vdash & \tau_5 \subseteq \tau_3 \prec N_1 \\
\Gamma; \varphi \vdash & \tau_3 \cap \tau_5 ; \varphi \vdash \tau_4 \subseteq \tau_6 \prec N_2 \\
\vdash & \lambda \mathcal{f}_x : \tau_3 \to \tau_4 . \lambda \mathcal{x} : \tau_6 . (\text{let } y \mathcal{f}_x \mathcal{g}_x = \mathcal{N}_1 x \text{ in let } \mathcal{y}^x = \mathcal{f} x \text{ in } \mathcal{N}_2 y)
\end{align*}
\]
The following holds from the induction hypothesis for some variables $a$ and $b$ that does not occur in $\tau_3$ and $\tau_6$ respectively.

\[
\text{Base}(\Gamma); \varphi \vdash N_1 : a : \tau_3 \rightarrow self(\tau_3, a) \quad (3)
\]

\[
\text{Base}(\Gamma, x : \tau_3 \sqcap \tau_5); \varphi \vdash N_2 : b : \tau_4 \rightarrow self(\tau_6, b) \quad (4)
\]

The following proof tree concludes the proof ($\Gamma' : \text{Base}(\Gamma), f : (x : \tau_3 \rightarrow \tau_4), x : \tau_3$).

\[
\begin{array}{c}
\Pi_1:
\Gamma'; \varphi \vdash N_1 x : self(\tau_3, x) \\
\Gamma'; y : \text{self}(\tau_3, x); \varphi \vdash (\text{let } z \tau_4 = f y \text{ in } N_2 z) : \tau_6 \\
\Gamma'; \text{self}(\tau_3, x); \varphi \vdash f y : \tau_4 \\
\end{array}
\]

\[
\begin{array}{c}
\Pi_2:
\Gamma'; \varphi \vdash N_1 x : self(\tau_3, x) \\
\Gamma'; \varphi \vdash x : \tau_5 \\
\Gamma'; \text{self}(\tau_3, x); \varphi \vdash (\text{let } z \tau_4 = f y \text{ in } N_2 z) : \tau_6 \\
\end{array}
\]

\[
\begin{array}{c}
\Pi_3:
\Gamma'; \varphi \vdash f : x : \tau_3 \rightarrow \tau_4 \\
\Gamma'; y : \text{self}(\tau_3, x); \varphi \vdash y : \tau_3 \\
\Gamma'; y : \text{self}(\tau_3, x); \varphi \vdash \frac{y}{x} : \tau_4 \\
\Gamma'; \text{self}(\tau_3, x); \varphi \vdash \frac{y}{x} : \tau_4 < : \tau_4 \\
\end{array}
\]

In $\Pi_3$, one of the leaves $\Gamma', y : \text{self}(\tau_3, x); \varphi \vdash \frac{y}{x} : \tau_4 < : \tau_4$ is proven as follows:

- If $\tau_3$ is a function type, so is $\tau_5$, and $x$ is a variable of function type.
  Therefore, $x$ does not occur in $\tau_4$, and $\frac{y}{x} : \tau_4 = \tau_4$ holds.
- If $\tau_3$ is a base type, $\Phi(\Gamma', y : \text{self}(\tau_3, x))$ implies $y = x$. Therefore, $\Gamma', y : \text{self}(\tau_3, x); \varphi \vdash \frac{y}{x} : \tau_4 < : \tau_4$ holds.

Also, in $\Pi_2$, one of the leaves $\Gamma', y : \text{self}(\tau_3, x), z : \tau_4; \varphi \vdash N_2 : b : \tau_4 \rightarrow self(\tau_6, b)$ is proven as follows:

- If $\tau_3$ is a function type, so is $\tau_3 \sqcap \tau_5$, and (4) is equivalent to $\text{Base}(\Gamma); \varphi \vdash N_2 : b : \tau_4 \rightarrow self(\tau_6, b)$. Therefore, (5) holds trivially.
• If \( \tau_3 \) is a base type, so is \( \tau_5 \). Let \( \tau_3 \equiv \{ x : B \mid \varphi_3 \} \) and \( \tau_5 \equiv \{ x : B \mid \varphi_5 \} \). Then, \( \Phi(\Gamma', y: \text{self}(\tau_3, x)) \) is logically equivalent to \( \Phi(\text{Base}(\Gamma, x: \tau_3 \cap \tau_5)) \) as follows:

\[
\Phi(\Gamma', y: \text{self}(\tau_3, x)) = \Phi(\text{Base}(\Gamma)) \land \varphi_5 \land [y/x] \varphi_3 \land y = x \\
\Phi(\text{Base}(\Gamma, x: \tau_3 \cap \tau_5)) = \Phi(\text{Base}(\Gamma)) \land \varphi_3 \land \varphi_5
\]

Therefore, \( \Gamma', y: \text{self}(\tau_3, x); \varphi \vdash N_2 : b; \tau_4 \rightarrow \text{self}(\tau_0, b) \) holds, and so does (5).

**Lemma 7.** (Lemma 2 in paper) \( \Gamma; \varphi \vdash M \leadsto N : \tau \) implies \( \Gamma; \varphi \vdash N : \tau \).

**Proof.** By induction on the derivation of \( \Gamma; \varphi \vdash M \leadsto N : \tau \).

- Case (CI-App).

\[
\Gamma; \varphi \vdash M_1 \leadsto N_1 : y: \tau_1 \rightarrow \tau_2 \quad \Gamma(x) = \tau_3 \quad \Gamma; \varphi \vdash \tau_3 \leq \tau_1 \leadsto N_2
\]

\[
\Gamma; \varphi \vdash M_1 x \leadsto (\text{let } x^{\tau_1} = N_2 x \text{ in } N_1 x) : [x/y] \tau_2
\]

From the induction hypothesis and Lemma 1, we obtain the following for a fresh \( z \).

\[
\Gamma; \varphi \vdash N_1 : y: \tau_1 \rightarrow \tau_2 \quad \Gamma; \varphi \vdash N_2 : z: \tau_3 \rightarrow \tau_1
\]

Noting \( [x/z] \tau_1 = \tau_1 \), it follows that \( \Gamma; \varphi \vdash N_2 x : \tau_1 \) holds, and \( \Gamma; \varphi \vdash (\text{let } x^{\tau_1} = N_2 x \text{ in } N_1 x) : [x/y] \tau_2 \) is obtained as expected.

- Case (CI-SUB).

\[
\Gamma; \varphi \vdash M_1 \leadsto N_1 : \tau_1 \quad \Gamma; \varphi \vdash \tau_1 \leq \tau \leadsto N_2
\]

\[
\Gamma; \varphi \vdash M_1 \leadsto \text{let } x^{\tau_1} = N_1 \text{ in } N_2 x : \tau
\]

From the induction hypothesis and Lemma 1, we obtain \( \Gamma; \varphi \vdash N_1 : \tau_1 \) and \( \Gamma; \varphi \vdash N_2 : y: \tau_1 \rightarrow \tau \) for a fresh variable \( y \). Since it is safe to assume that \( x \) does not appear in \( N_2 \) and \( y \neq x \), \( \Gamma, x : \tau_1; \varphi \vdash N_2 : y: \tau_1 \rightarrow \tau \) also holds. Thus, \( \Gamma, x : \tau_1; \varphi \vdash N_2 x : \tau \) holds, and hence we obtain the result as follows (note \( [x/y] \tau = \tau \)).

\[
\Gamma; \varphi \vdash N_1 : \tau_1 \quad \Gamma, x : \tau_1; \varphi \vdash N_2 : y: \tau_1 \rightarrow \tau \quad \Gamma, x : \tau_1; \varphi \vdash x : \tau_1
\]

\[
\Gamma; \varphi \vdash \text{let } x^{\tau_1} = N_1 \text{ in } N_2 x : \tau
\]

- Other cases are trivial.

**Definition 1.** Substitution of type \([v/x]\tau\) and type environment \([v/x]\Gamma\) is defined in Figure 17.

**Lemma 8.** \( \Gamma; \varphi \vdash \tau <: \tau' \) and \( \vdash v : \Gamma(x) \) imply \([v/x]\Gamma; [v/x]\varphi \vdash [v/x]\tau <: [v/x]\tau' \).

**Proof.** By induction on \( \Gamma; \varphi \vdash \tau <: \tau' \).
Proof. By induction on Lemma 9.

\[ \text{[v/x]}\tau \]
\[ \text{[v/x]Γ} \]
\[ [v/x]\{y : B \mid \varphi\} = \{y : B \mid [v/x]\varphi\} \]
\[ [v/x]\varphi = \varnothing \]
\[ [v/x](\tau_1 \rightarrow \tau_2) = y : ([v/x]\tau_1) \rightarrow ([v/x]\tau_2) \quad [v/x](\Gamma, x : \tau) = [v/x]\Gamma \]
\[ [v/x](\Gamma, y : \tau) = ([v/x]\Gamma), y : [v/x]\tau \quad (x \neq y) \]

Fig. 17. Substitution of type and type environment.

- Case (Sub-Base).

\[ \Gamma; \varphi \vdash [v/x]\tau_3 \quad \Gamma; y : [v/x]\tau_1 \quad \Gamma; y : [v/x]\tau_2 \]
\[ \Gamma; y : [v/x]\tau_3 ; \varphi \vdash [v/x]\tau_2 \quad \Gamma; y : [v/x]\tau_2 \]
\[ \Gamma; y : [v/x]\tau_3 \rightarrow [v/x]\tau_2 \]

We can assume \( x \neq y \) w.l.o.g by alpha renaming. From the induction hypothesis, we obtain:

\[ [v/x]\Gamma; [v/x]\varphi \vdash [v/x]\varphi_1 \]
\[ ([v/x]\Gamma), x : [v/x]\tau_3 ; [v/x]\varphi \vdash [v/x]\tau_2 \]
\[ ([v/x]\Gamma), y : [v/x]\tau_3 ; [v/x]\varphi \vdash [v/x]\tau_4 \]

Therefore, \([v/x]\Gamma; [v/x]\varphi \vdash [v/x]((y : \tau_1 \rightarrow \tau_2) \rightarrow [v/x]((y : \tau_3 \rightarrow \tau_4)) \)
holds as expected.

Lemma 9. \( \Gamma; \varphi \vdash N : \tau \) and \( \vdash v : \Gamma(x) \) imply \([v/x]\Gamma; [v/x]\varphi \vdash [v/x]N : [v/x]\tau \).

Proof. By induction on \( \Gamma; \varphi \vdash N : \tau \).

- Case (CT-Con), (CT-VF) and (CT-VB) is trivial.

- Case (CT-Lam).

\[ \Gamma; y : \tau_1 ; \varphi \vdash N_1 : \tau_2 \]
\[ \Gamma; \varphi \vdash \lambda y^{\tau_1} N_1 : y : \tau_1 \rightarrow \tau_2 \]

We can assume \( x \neq y \) by alpha renaming. From the induction hypothesis,

\[ ([v/x]\Gamma), y : [v/x]\tau_1 ; [v/x]\varphi \vdash [v/x]N_1 : [v/x]\tau_2 \]
holds. Therefore, \([v/x]\Gamma; [v/x]\varphi \vdash \lambda y^{[v/x]\tau_1} \tau_1 . [v/x]N_1 : y : ([v/x]\tau_1) \rightarrow [v/x]\tau_2 \]
holds as expected.
– Case (CT-Fix) is similar to the case (CT-LAM).
– Case (CT-App).

\[
\begin{array}{c}
\Gamma; \varphi \vdash N_1 : \tau_1 \rightarrow \tau_2 \\
\Gamma; \varphi \vdash v_1 : \tau_1
\end{array}
\quad
\begin{array}{c}
\Gamma; \varphi \vdash N_1 v_1 : [v_1/y] \tau_2
\end{array}
\]

We assume \(x \neq y\) by alpha renaming. From the induction hypothesis, we obtain the following.

\[
[v/x] \Gamma; [v/x] \varphi \vdash [v/x] N_1 : [v/x] \tau_1 \rightarrow [v/x] \tau_2
\]

\[
[v/x] \Gamma; [v/x] \varphi \vdash [v/x] v_1 : [v/x] \tau_1
\]

Therefore, \([v/x] \Gamma; [v/x] \varphi \vdash ([v/x] N_1)([v/x] v_1) : [[v/x] v_1/y][v/x] \tau_2\) holds as expected.

– Case (CT-Sub).

\[
\begin{array}{c}
\Gamma; \varphi \vdash N : \tau \\
\Gamma; \varphi \vdash \tau' \triangleleft \tau
\end{array}
\]

\[
\begin{array}{c}
\Gamma; \varphi \vdash N : \tau
\end{array}
\]

From the induction hypothesis and Lemma 8, we obtain the following.

\[
[v/x] \Gamma; [v/x] \varphi \vdash [v/x] N : [v/x] \tau'
\]

\[
[v/x] \Gamma; [v/x] \varphi \vdash [v/x] \tau' \triangleleft : [v/x] \tau
\]

Therefore, \([v/x] \Gamma; [v/x] \varphi \vdash [v/x] N : [v/x] \tau\) follows as expected.

– Other cases are trivial from the induction hypothesis.

From now on, we abbreviate \(\emptyset; \text{true} \vdash N : \tau\) as \(\vdash N : \tau\).

**Lemma 10 (Progress).** Suppose \(\vdash N : \tau\) holds, and \(\text{ev}(c, v)\) is defined whenever \(\vdash c v : \tau\) holds for some \(\tau\). Then, one of the following holds.

– \(N\) is a value.
– \(N \rightarrow N'\) for some \(N'\)
– \(N \rightarrow \text{error}\)

**Proof.** By induction on the derivation of \(\vdash N : \tau\).

– In cases (CT-Con), (CT-VF), (CT-VB), (CT-LAM) and (CT-Fix), \(N\) is a value.

– Case (CT-App).

\[
\begin{array}{c}
\vdash N_1 : \tau_1 \rightarrow \tau_2 \\
\vdash v : \tau_1
\end{array}
\quad
\begin{array}{c}
\vdash N_1 v : [v/x] \tau_2
\end{array}
\]

We split cases from the induction hypothesis.

• If \(N_1\) is a value, then \(N_1\) is either a constant, a lambda term, or a recursive function. If \(N_1\) is a constant \(c\), then \(c v \rightarrow \text{ev}(c, v)\) holds since \(\text{ev}(c, v)\) is defined whenever \(c v\) is well-typed. If \(N_1\) is a lambda term or a recursive function, there exists \(N'\) such that \(N_1 v \rightarrow N'\) holds.
Proof. By induction on $c, v$ and $\text{Supp}(\text{Lemma 11 (Preservation). Suppose } \vdash cv : \tau \text{ implies } \vdash cv(c, v) : \tau \text{ for every } c, v \text{ and } \tau. \text{ Then, } \vdash N : \tau \text{ and } N \rightarrow N' \text{ imply } \vdash N' : \tau.}

Proof. By induction on $N \rightarrow N'$.

- Case $\text{assert} \,(\text{true}) ; N_1 \rightarrow N_1$. From the inversion of $\vdash \text{assert} \,(\text{true}) ; N_1 : \tau$, it follows that $\vdash N_1 : \tau$.
- Case $\lambda x^{\tau_1}. N_1 \rightarrow [v/x]N_1$. The derivation of $\vdash (\lambda x^{\tau_1}. N_1) v : \tau$ must be of the following form.

Using Lemma 9, $x : \tau_1 ; \text{true} \vdash [v/x]N_1 : [v/x]\tau_2$ holds. Since $x$ does not appear freely in $[v/x]N_1$, this entails $\vdash [v/x]N_1 : [v/x]\tau_2$ as expected.
Proof. From Lemma 13.

\[ f : (x : \tau_2 \to \tau_3), x : \tau_2 ; \text{true} \vdash N_1 : \tau_3 \]

\[ \vdash v : \tau_2 \]

\[ \vdash \text{fix}(f^{x \to \tau_2 \to \tau_3}, x, N_1) : x : \tau_2 \to \tau_3 \]

Using Lemma 9, \( f : (x : \tau_2 \to \tau_3), x : \tau_2 ; \text{true} \vdash [v/x, \text{fix}(f^{x \to \tau_2 \to \tau_3}, x, N_1)] : \tau_3 \) holds. Therefore, \( \vdash [v/x, \text{fix}(f^{x \to \tau_2 \to \tau_3}, x, N_1)] : \tau_3 \) holds as expected.

- Case \( c v \to ev(c, v) \) follows from the assumption.
- Case if true then \( N_1 \) else \( N_2 \to N_1 \) is trivial.
- Case if false then \( N_1 \) else \( N_2 \to N_2 \) is trivial.
- Other cases are trivial.

**Theorem 4 (Type Safety).** Suppose \( \vdash c : \tau \) implies \( \vdash ev(c, v) : \tau \) for every \( c, v \) and \( \tau \). Then, \( \emptyset ; \text{true} \vdash M \leadsto N : \tau \) implies either \( N \leadsto^* v, N \uparrow \) or \( N \leadsto^* \text{error} \).

**Proof.** From Lemma 7, \( \vdash N : \tau \) holds. Therefore, the result is obtained from Lemma 10 and Lemma 11.

### B.5 Properties about Precision

**Lemma 12.** \( x \vdash \tau_1 \subseteq \tau_2 \) and \( \bar{x} \vdash \tau_3 \subseteq \tau_4 \) imply \( \bar{x} \vdash \tau_1 \cap \tau_3 \subseteq \tau_2 \cap \tau_4 \).

**Lemma 13.** (Lemma 3 in paper) \( \Gamma ; \varphi \vdash \tau_1 \subseteq \tau_2 \), \( \text{dom}(\Gamma) \vdash \tau_1 \subseteq \tau_3 \), \( \text{dom}(\Gamma) \vdash \tau_2 \subseteq \tau_4 \) and \( \Gamma \subseteq \Gamma' \) imply \( \Gamma' ; \varphi \vdash \tau_3 \subseteq \tau_4 \).

**Proof.** By induction on \( \Gamma ; \varphi \vdash \tau_1 \subseteq \tau_2 \).

- Case (Cast-Base). The derivation must be of the following form.

\[ \vdash \exists \text{BT}(\Gamma), x : B. \Phi(\Gamma) \land \varphi \land \varphi_1 \land \varphi_2 \]

\[ \vdash \forall \text{BT}(\Gamma), x : B. \Phi(\Gamma) \land \varphi \land \varphi_1 \Rightarrow (\varphi' \Leftrightarrow \varphi_2) \]

\[ \Gamma ; \varphi \vdash \{ x : B \mid \varphi_3 \} \subseteq \{ x : B \mid \varphi_2 \} \Rightarrow \lambda x[x : B[x_1]]. \text{assert}(\varphi') ; x \]

Let \( \varphi_3 \) such that \( \tau_3 \equiv \{ x : B \mid \varphi_3 \} \), \( \varphi_4 \) such that \( \tau_4 \equiv \{ x : B \mid \varphi_4 \} \), and \( \Gamma' \) such that \( \Gamma \subseteq \Gamma' \). From \( \text{dom}(\Gamma) \vdash \tau_1 \subseteq \tau_3 \), \( \text{dom}(\Gamma) \vdash \tau_2 \subseteq \tau_4 \) and \( \Gamma \subseteq \Gamma' \), we obtain the following, noting that only a variable of base type appears in the predicates.

\[ \forall \text{BT}(\Gamma), x : B. \Phi_1 \Rightarrow \varphi_3 \]

\[ \forall \text{BT}(\Gamma), x : B. \Phi_2 \Rightarrow \varphi_4 \]

\[ \forall \text{BT}(\Gamma). \Phi(\Gamma) \Rightarrow \Phi(\Gamma') \]

Therefore, \( \exists \text{BT}(\Gamma'), x : B. \Phi(\Gamma') \land \varphi \land \varphi_3 \land \varphi_4 \) holds, noting \( \text{BT}(\Gamma) = \text{BT}(\Gamma') \).

Let \( \varphi'' \) such that \( \forall \text{BT}(\Gamma'), x : B. \Phi(\Gamma') \land \varphi \land \varphi_3 \Rightarrow (\varphi'' \Leftrightarrow \varphi_4) \). Thus, \( \Gamma' ; \varphi \vdash \{ x : B \mid \varphi_3 \} \subseteq \{ x : B \mid \varphi_4 \} \Rightarrow \lambda x[x : B[x_3]]. \text{assert}(\varphi'') ; x \) holds as expected.
– Case (Cast-Fun). The derivation must be of the following form.

\[
\Gamma; \varphi \vdash \tau_2' \preceq \tau_1' \leadsto N_1 \quad \Gamma; \varphi \vdash \tau_1' \sqcap \tau_2''; \varphi \vdash \tau_2'' \preceq \tau_1'' \leadsto N_2
\]

\[
\sim \lambda f^{x: \tau_1' \rightarrow \tau_2''}. \lambda x^{\tau_2''}. (\textnormal{let } x^{\tau_1' \sqcap \tau_2''} = N_1 \textnormal{ in let } y^{\tau_2''} = f \textnormal{ in } N_2 \textnormal{ y})
\]

Let \( \tau_3', \tau_3'', \tau_4' \) and \( \tau_4'' \) such that \( \tau_3 \equiv x: \tau_3' \rightarrow \tau_3'' \) and \( \tau_4 \equiv x: \tau_4' \rightarrow \tau_4'' \). From \( \text{dom}(\Gamma) \vdash \tau_1 \sqsubseteq \tau_3 \) and \( \text{dom}(\Gamma) \vdash \tau_2 \sqsubseteq \tau_4 \), the following holds.

\[
\text{dom}(\Gamma) \vdash \tau_1' \sqsubseteq \tau_3' \quad \text{dom}(\Gamma) \vdash \tau_1'' \sqsubseteq \tau_3''
\]

Using Lemma 12, \( \text{dom}(\Gamma) \vdash \tau_1' \sqcap \tau_2' \sqsubseteq \tau_3' \sqcap \tau_4' \) holds. Let \( \Gamma' \) such that \( \Gamma \sqsubseteq \Gamma' \). Then, \( \Gamma; x: \tau_1' \sqcap \tau_2' \sqsubseteq \Gamma', x: \tau_3' \sqcap \tau_4' \) holds. From the induction hypothesis, \( \Gamma'; \varphi \vdash \tau_3' \preceq \tau_3 \leadsto N_3 \) and \( \Gamma'; x: \tau_3' \sqcap \tau_4'; \varphi \vdash \tau_3'' \sqsubseteq \tau_3'' \leadsto N_4 \) hold for some \( N_3 \) and \( N_4 \). Therefore, we obtain \( \Gamma; \varphi \vdash x: \tau_3' \rightarrow \tau_3'' \sqsubseteq x: \tau_4' \rightarrow \tau_4'' \leadsto \lambda f^{x: \tau_3' \rightarrow \tau_3''}. \lambda y^{\tau_4''}. (\text{let } x^{\tau_3' \sqcap \tau_4''} = N_3 \text{ in let } y^{\tau_4''} = f \text{ in } N_4 \text{ y}) \) as expected.

**Lemma 14.** (Lemma 4 in paper) Suppose \( \Gamma \sqsubseteq \Gamma', \text{dom}(\Gamma) \vdash \tau_1 \sqsubseteq \tau_1' \) and \( \text{dom}(\Gamma) \vdash \tau_2 \sqsubseteq \tau_2' \). Then, \( \Gamma; \varphi \vdash \tau_2 \preceq N \) and \( \Gamma; \varphi \vdash \tau_2' \sqsubseteq \tau_2' \leadsto N' \) implies \( \Gamma; \varphi \vdash N \sqsubseteq N' \).

**Proof.** By induction on the derivation of \( \Gamma; \varphi \vdash \tau_2 \preceq N \).

– Case (Cast-Base).

\[
\Gamma; \varphi \vdash \exists B. \Phi(\Gamma) \land \varphi \land \varphi_1 \land \varphi_2
\]

\[
\sim \forall B. \Phi(\Gamma) \land \varphi \land \varphi_1 \Rightarrow (\varphi' \iff \varphi_2)
\]

\[
\text{assert}(\varphi'); x
\]

From the inversion of \( \text{dom}(\Gamma) \vdash \tau_1 \sqsubseteq \tau_1' \) and \( \text{dom}(\Gamma) \vdash \tau_2 \sqsubseteq \tau_2' \), it must be that \( \tau_1' \equiv \{ x: B \mid \varphi_1' \} \) and \( \tau_2' \equiv \{ x: B \mid \varphi_2' \} \) for some \( \varphi_1' \) and \( \varphi_2' \) where the following holds.

\[
\forall \text{dom}(\Gamma), x. \varphi_1 \Rightarrow \varphi_1'
\]

\[
\forall \text{dom}(\Gamma), x. \varphi_2 \Rightarrow \varphi_2'
\]

Since the variables of function types cannot be used in the predicates, we can restrict the quantified variables to \( \text{dom}(\Phi(\Gamma)) \) as follows.

\[
\forall \text{dom}(\Phi(\Gamma)), x. \varphi_1 \Rightarrow \varphi_1' \quad (6)
\]

\[
\forall \text{dom}(\Phi(\Gamma)), x. \varphi_2 \Rightarrow \varphi_2' \quad (7)
\]

The derivation of \( \Gamma; \varphi \vdash \tau_1' \sqsubseteq \tau_2' \leadsto N' \) must be of the following form.

\[
\Gamma; \varphi \vdash \exists B. \Phi(\Gamma) \land \varphi \land \varphi_1' \land \varphi_2'
\]

\[
\sim \forall B. \Phi(\Gamma) \land \varphi \land \varphi_1' \Rightarrow (\varphi'' \iff \varphi_2')
\]

\[
\text{assert}(\varphi'''); x
\]

Therefore, we obtain \( \forall B. \Phi(\Gamma) \land \varphi \land \varphi_1 \land \varphi' \Rightarrow \varphi'' \) as follows: Assume \( \Phi(\Gamma) \land \varphi \land \varphi_1 \land \varphi' \). Then, \( \varphi_1' \) follows from (6), and \( \varphi_2' \) follows from the premise.
of \( \Gamma; \varphi \vdash \tau_1 \subseteq \tau_2 \leadsto N \). Therefore, \( \varphi'_2 \) is obtained by (7). With the premise of \( \Gamma; \varphi \vdash \tau_1' \subseteq \tau_2' \leadsto N' \), we obtain \( \varphi'' \) as expected. Thus, we obtain \( \Gamma; \varphi \vdash N \sqsubseteq N' \) as follows.

\[
\forall \text{BT}(I), x : B. \Phi(I) \wedge \varphi \wedge \varphi_1 \wedge \varphi' \Rightarrow \varphi''
\]

\[
\Gamma; x : \{ x : B \mid \varphi_1 \}; \varphi \vdash \text{assert}(\varphi') ; x \subseteq \text{assert}(\varphi'') ; x
\]

\[
\Gamma; \varphi \vdash \lambda x \{ x : B \mid \varphi_1 \}. \text{assert}(\varphi') ; x \subseteq \lambda x \{ x : B \mid \varphi_1 \}. \text{assert}(\varphi'') ; x
\]

- Case (Cast-Fun).

\[
\Gamma; \varphi \vdash \tau_5 \subseteq \tau_3 \leadsto N_1 \quad \Gamma; x : \tau_3 \sqcap \tau_5 ; \varphi \vdash \tau_4 \subseteq \tau_6 \leadsto N_2
\]

\[
\sim \lambda f : \tau_3 \rightarrow \tau_4, \lambda x : \tau_5. (\text{let } x \tau_3 = N_1 x \text{ in let } y^4 = f x \text{ in } N_2 y)
\]

From the inversion of \( \text{dom}(\Gamma) \vdash \tau_1 \subseteq \tau'_1 \) and \( \text{dom}(\Gamma) \vdash \tau_2 \sqsubseteq \tau'_2 \), it must be that \( \tau'_1 \equiv x : \tau_3 \rightarrow \tau'_4 \) and \( \tau'_2 \equiv x : \tau_5 \rightarrow \tau'_6 \) for some \( \tau'_1, \tau'_4, \tau'_5, \tau'_6 \) where the following holds.

\[
dom(\Gamma) \vdash \tau_3 \subseteq \tau'_3 \quad (8)
\]

\[
dom(\Gamma) \vdash \tau_5 \subseteq \tau'_5 \quad (9)
\]

\[
dom(\Gamma), x \vdash \tau_4 \sqsubseteq \tau'_4 \quad (10)
\]

\[
dom(\Gamma), x \vdash \tau_6 \sqsubseteq \tau'_6 \quad (11)
\]

The derivation of \( \Gamma'; \varphi \vdash \tau'_1 \subseteq \tau'_2 \leadsto N' \) must be of the following form.

\[
\Gamma'; \varphi \vdash \tau'_5 \subseteq \tau'_3 \leadsto N'_1 \quad \Gamma', x : \tau'_3 \sqcap \tau'_5 ; \varphi \vdash \tau'_4 \subseteq \tau'_6 \leadsto N'_2
\]

\[
\sim \lambda f : \tau'_3 \rightarrow \tau'_4, \lambda x : \tau'_5. (\text{let } x \tau'_3 = N'_1 x \text{ in let } y^4 = f x \text{ in } N'_2 y)
\]

With (8) and (9), we obtain \( \Gamma; \varphi \vdash N_1 \sqsubseteq N'_1 \) from the induction hypothesis. Noting \( \Gamma, x : \tau_3 \sqcap \tau_5 \sqsubseteq \Gamma', x : \tau'_3 \sqcap \tau'_5 \), we obtain \( \Gamma, x : \tau_3 \sqcap \tau_5 ; \varphi \vdash N_2 \sqsubseteq N'_2 \) from the induction hypothesis with (10) and (11). Thus, \( \Gamma; \varphi \vdash N \sqsubseteq N' \) follows as expected.

### B.6 Gradual Guarantee

Figure 18 and Figure 19 presents the full definition of the precision of terms \( \bar{x} \vdash M_1 \sqsubseteq M_2 \) and the precision of cast terms \( \Gamma; \varphi \vdash N_1 \sqsubseteq N_2 \) respectively.

**Remark 2.** The notation \( \bar{x} \setminus \{ y \} \) denotes a sequence of variables that is created by removing \( y \) from \( \bar{x} \).

**Lemma 15.** \( \text{dom}(\Gamma) \vdash \tau_1 \subseteq \tau_2, v_1 \subseteq v_2 \) and \( \vdash v_1 : \Gamma(x) \) imply \( \text{dom}(\Gamma) \setminus \{ x \}) \vdash [v_1/x]\tau_1 \subseteq [v_2/x]\tau_2 \).

**Proof.** By induction on \( \Gamma \vdash \tau_1 \subseteq \tau_2 \).
\[ x \vdash M_1 \subseteq M_2 \]

\[ \Gamma; \varphi \vdash c \subseteq c \quad \text{(PC-CONST)} \quad \Gamma; \varphi \vdash x \subseteq x \quad \text{(PC-VAR)} \]

\[ \bar{y} \vdash M_1 \subseteq M_2 \quad \Gamma; \varphi \vdash \lambda x : \tau_1 . M_1 \subseteq \lambda x : \tau_2 . M_2 \quad \text{(PM-LAM)} \]

\[ \bar{y} \vdash \lambda x : \tau_1 . M_1 \subseteq \lambda x : \tau_2 . M_2 \quad \text{(PM-ANNOT)} \]

\[ \bar{y} \vdash (M_1 : \tau_1) \subseteq (M_2 : \tau_2) \quad \text{(PM-APP)} \]

\[ \bar{y} \vdash (let \ x = M_1 \ in \ M_2) \subseteq (let \ x = M_1' \ in \ M_2') \quad \text{(PM-LET)} \]

\[ \bar{y} \vdash \text{fix}(f : (x : \tau_1 \rightarrow \tau_2), x, M) \subseteq \text{fix}(f : (x : \tau_3 \rightarrow \tau_4), x, M') \quad \text{(PM-FIX)} \]

\[ \bar{y} \vdash (if \ x \ then \ M_1 \ else \ M_2) \subseteq (if \ x \ then \ M_1' \ else \ M_2') \quad \text{(PM-IF)} \]

**Fig. 18.** Precision relation over terms (full version of Figure 14).

\[
\begin{array}{c}
\dfrac{
\Gamma; \varphi \vdash N_1 \subseteq N_2
}{
\Gamma; \varphi \vdash c \subseteq c \quad \text{(PC-CONST)}
\Gamma; \varphi \vdash x \subseteq x \quad \text{(PC-VAR)}
\dfrac{
\text{dom}(\Gamma) \vdash \tau_1 \subseteq \tau_2 \\
\Gamma; x : \tau_1 ; \varphi \vdash N_1 \subseteq N_2 \\
\varnothing; \text{true} \vdash v_1 \subseteq v_2
}{
\Gamma; \varphi \vdash \lambda x : \tau_1 . N_1 \subseteq \lambda x : \tau_2 . N_2 \\
\text{(PC-LAM)}
\Gamma; \varphi \vdash N_1 \subseteq N_2
ty_{1} \vdash \varphi \vdash N_2 \subseteq N_3
\dfrac{
\Gamma; \varphi \vdash x \vdash \lambda x : \tau_1 \rightarrow \tau_2 . f \vdash x : \tau_1 \rightarrow \tau_2 . x : \tau_1 ; \varphi \vdash N_3 \subseteq N_2
}{
\Gamma; \varphi \vdash \text{fix}(f^{x : \tau_1 \rightarrow \tau_2}, x, N_1) \subseteq \text{fix}(f^{x : \tau_3 \rightarrow \tau_2}, x, N_2) \\
\text{(PC-APP)}
\dfrac{
\text{dom}(\Gamma) \vdash x : \tau_1 \rightarrow \tau_2 \subseteq x : \tau_1 \rightarrow \tau_2' \\
\Gamma; \varphi \vdash N_1 \subseteq N_2
}{
\Gamma; \varphi \vdash \text{fix}(f^{x : \tau_1 \rightarrow \tau_2}, x, N_1) \subseteq \text{fix}(f^{x : \tau_1 \rightarrow \tau_2'}, x, N_2) \\
\text{(PC-APP)}
\dfrac{
\Gamma; \varphi \vdash N_1 \subseteq N_3 \\
\Gamma; x : \tau_1 ; \varphi \vdash N_2 \subseteq N_4
}{
\Gamma; \varphi \vdash \text{let} \ x = N_1 \ in \ N_2 \subseteq \text{let} \ x = N_3 \ in \ N_4 \\
\text{(PC-LAM)}
\varnothing; \text{true} \vdash v_1 \subseteq v_2 \\
\Gamma; \varphi \vdash \varphi \land v_1 \vdash N_1 \subseteq N_3 \\
\Gamma; \varphi \vdash \neg v_1 \vdash N_2 \subseteq N_4
}{
\Gamma; \varphi \vdash \text{if} \ v_1 \ then \ N_1 \ else \ N_2 \subseteq \text{if} \ v_2 \ then \ N_3 \ else \ N_4 \\
\text{(PC-APP)}
\dfrac{
\forall \Gamma; \Phi(\Gamma) \land \varphi \land \varphi_1 \Rightarrow \varphi_2 \\
\Gamma; \varphi \vdash \varphi_1 \vdash N_1 \subseteq N_2
}{
\Gamma; \varphi \vdash \text{assert}(\varphi_1) ; N_1 \subseteq \text{assert}(\varphi_2) ; N_2 \\
\text{(PC-ASSERT)}
\}
\end{array}
\]

**Fig. 19.** Precision relation of the cast terms (full version of Figure 14).
Case (Prec-Base).

\[
\forall \text{dom}(\Gamma), y : \varphi_1 \Rightarrow \varphi_2 \\
\text{dom}(\Gamma) \vdash \{ y : B \mid \varphi_1 \} \subseteq \{ y : B \mid \varphi_2 \}
\]

We can assume \( y \neq x \) w.l.o.g by alpha renaming.

If \( \Gamma(x) \) is a function type, \( x \) does not appear in \( \varphi_1 \) or \( \varphi_2 \). Therefore, the result follows trivially.

If \( \Gamma(x) \) is a base type, \( v_1 \) is a constant, and thus \( v_1 = v_2 \) follows from \( v_1 \subseteq v_2 \).

Therefore, \( \forall \text{dom}(\Gamma) \setminus \{ x \}, y, [v_1/x]\varphi_1 \Rightarrow [v_2/x]\varphi_2 \) follows.

Case (Prec-Fun).

\[
\begin{align*}
\text{dom}(\Gamma) &\vdash \tau_3 \subseteq \tau_5 & \text{dom}(\Gamma) &\vdash \tau_4 \subseteq \tau_6 \\
\therefore \text{dom}(\Gamma) &\vdash y_3 : \tau_3 \rightarrow \tau_4 \subseteq y_5 : \tau_5 \rightarrow \tau_6
\end{align*}
\]

We can assume \( y \neq x \) w.l.o.g by alpha renaming. From the induction hypothesis, the following holds.

\[
\text{dom}(\Gamma) \setminus \{ x \} \vdash [v_1/x]\tau_3 \subseteq [v_2/x]\tau_5 \\
(\text{dom}(\Gamma) \setminus \{ x \}), y \vdash [v_1/x]\tau_3 \subseteq [v_2/x]\tau_5
\]

Therefore, we obtain \( \text{dom}(\Gamma) \setminus \{ x \}, y, [v_1/x](y_3 : \tau_3 \rightarrow \tau_4) \subseteq [v_2/x](y_5 : \tau_5 \rightarrow \tau_6) \) as expected.

**Lemma 16.** \( \Gamma; \varphi \vdash N \subseteq N' \), \( v \subseteq v' \) and \( v : \Gamma(x) \) imply \( [v/x] \Gamma; [v/x] \varphi \vdash [v/x] N \subseteq [v'/x] N' \).

**Proof.** By induction on \( \Gamma; \varphi \vdash N \subseteq N' \):

- Case (PC-Const) is trivial.
- Case (PC-Var). If \( N = N' = x \), then \([v/x] N = v \) and \([v'/x] N' = v' \).

Therefore, \([v/x] \Gamma; [v/x] \varphi \vdash [v/x] N \subseteq [v'/x] N' \) holds from \( v \subseteq v' \). Otherwise, \([v/x] \Gamma; [v/x] \varphi \vdash [v/x] N \subseteq [v'/x] N' \) trivially holds from (PC-Var).
- Case (PC-Lam).

\[
\begin{align*}
\text{dom}(\Gamma) &\vdash \tau_1 \subseteq \tau_2 & \Gamma, y : \tau_1; \varphi &\vdash N_1 \subseteq N_2 \\
\Gamma &\vdash \lambda y \tau_1 : N_1 \subseteq \lambda y \tau_2 : N_2
\end{align*}
\]

We can assume \( x \neq y \) w.l.o.g by alpha renaming. From the induction hypothesis, \(([v/x] \Gamma), y : [v/x] \tau_1; [v/x] \varphi \vdash [v/x] N_1 \subseteq [v'/x] N_2 \) holds. Also, \( \text{dom}(\Gamma) \setminus \{ x \} \vdash [v/x] \tau_1 \subseteq [v'/x] \tau_2 \) holds from Lemma 15.

Noting \( \text{dom}(\Gamma) = \text{dom}(\Gamma) \setminus \{ x \} \), we obtain \([v/x] \Gamma; [v/x] \varphi \vdash \lambda y [v/x] \tau_1; [v/x] N_1 \subseteq \lambda y [v'/x] \tau_2; [v'/x] N_2 \) as expected.
- Case (PC-Let).

\[
\begin{align*}
\Gamma &\vdash N_1 \subseteq N_3 & \Gamma, y : \tau_1; \varphi &\vdash N_2 \subseteq N_4 \\
\therefore \Gamma; \varphi &\vdash \text{let} y \tau_1 = N_1 \text{ in } N_2 \subseteq \text{let} y \tau_1 = N_3 \text{ in } N_4
\end{align*}
\]
We can assume \( x \neq y \) w.l.o.g by alpha renaming. From the induction hypothesis, followings hold.
\[
[v/x] \Gamma; [v/x] \varphi \vdash [v/x] N_1 \sqsubseteq [v/x] N_3
\]
\(([v/x] \Gamma), y : [v/x] \tau_1; [v/x] \varphi \vdash [v/x] N_2 \sqsubseteq [v/x] N_4\)

Therefore,
\[
[v/x] \Gamma; [v/x] \varphi \vdash \text{let } y^{[v/x] \tau_1} = [v/x] N_1 \text{ in } [v/x] N_2 \sqsubseteq (\text{let } y^{[v/x] \tau_3} = [v/x] N_3 \text{ in } [v/x] N_4)
\]
holds as expected.

- Case (PC-Fix) and (PC-Let) is similar to (PC-Lam).
- Case (PC-App) and (PC-If) is immediate from the induction hypothesis.
- Case (PC-Assert).

\[
\forall \text{BT}(\Gamma), \Phi(\Gamma) \land \varphi \land \varphi_1 \Rightarrow \varphi_2 \quad \Gamma; \varphi \land \varphi_1 \vdash N_1 \sqsubseteq N_2
\]

From the induction hypothesis, \([v/x] \Gamma; [v/x] \varphi \land [v/x] \varphi_1 \vdash [v/x] N_1 \sqsubseteq [v'/x] N_2\) holds.

If \(\Gamma(x)\) is a function type, then \(\text{BT}([v/x] \Gamma) = \text{BT}(\Gamma)\) and \(\Phi([v/x] \Gamma) = \Phi(\Gamma)\) holds. Also, \(x\) does not appear in \(\varphi, \varphi_1\) or \(\varphi_2\). Therefore, \(\forall \text{BT}([v/x] \Gamma), \Phi([v/x] \Gamma) \land [v/x] \varphi \land [v/x] \varphi_1 \Rightarrow [v/x] \varphi_2\) holds.

If \(\Gamma(y)\) is a base type, \(v = v'\) holds from \(v \sqsubseteq v'\) since \(v\) must be a constant.

Also, \(\text{BT}([v/x] \Gamma) = \text{BT}(\Gamma) \setminus \{x\}\) and \(\Phi([v/x] \Gamma) = [v/x] \Phi(\Gamma)\) holds. Therefore, \(\forall \text{BT}([v/x] \Gamma), \Phi([v/x] \Gamma) \land [v/x] \varphi \land [v/x] \varphi_1 \rightarrow [v/x] \varphi_2\) holds.

Therefore, in either cases,
\[
\forall \text{BT}([v/x] \Gamma), \Phi([v/x] \Gamma) \land [v/x] \varphi \land [v/x] \varphi_1 \Rightarrow [v'/x] \varphi_2
\]
follows as expected.

From now on, we abbreviate \(\varnothing; \text{true} \vdash N_1 \sqsubseteq N_2\) as \(N_1 \sqsubseteq N_2\).

As we stated in the paper, we assume the following property for the primitive functions.

**Assumption 3** If \(\text{ev}(c, v_2)\) and \(\text{ev}(c, v_1)\) are both defined, then \(v_1 \sqsubseteq v_2\) implies \(\text{ev}(c, v_1) \sqsubseteq \text{ev}(c, v_2)\).

**Lemma 17.** (Proposition 2 in paper) Suppose \(\vdash N_1 : \tau\) and \(\vdash N_2 : \tau'\) for some \(\tau\) and \(\tau'\). Then, \(N_1 \sqsubseteq N_2\) and \(N_1 \rightarrow N_1'\) imply \(N_2 \rightarrow N_2'\) and \(N_1' \sqsubseteq N_2'\).

**Proof.** By induction on \(N_1 \rightarrow N_1'\).

- Case \(\text{assert}(true)\): \(N_3 \rightarrow N_3\). From the inversion of \(N_1 \sqsubseteq N_2\), it must be that \(N_2 \equiv \text{assert}(true)\); \(N_4\) for some \(N_4\) where \(N_3 \sqsubseteq N_4\). Therefore, \(\text{assert}(true)\); \(N_4 \rightarrow N_4\) holds.

- Case \((\lambda x^{\tau_1}. N_3) v_1 \rightarrow [v_1/x] N_3\). From the inversion of \(N_1 \sqsubseteq N_2\), it must be that \(N_2 \equiv (\lambda x^{\tau_2}. N_4) v_2\) for some \(\tau_2\), \(N_4\) and \(v_2\) where \(\tau_1 \sqsubseteq \tau_2\), \(v_1 \sqsubseteq v_2\) and \(x : \tau_1\). \(\text{true} \vdash N_3 \sqsubseteq N_4\). Also, from the inversion of \(\vdash (\lambda x^{\tau_1}. N_3) v_1 : \tau\), we obtain \(\vdash [v_1/x] N_3 \sqsubseteq [v_2/x] N_4\) holds as expected.
Case (fix(\(f^1, x, N_1\))v_1 \rightarrow \[v_1/x, \text{fix}(f^1, x, N_1)/f\]N_1. From the inversion of \(N_1 \subseteq N_2\), it must be that \(N_2 \equiv (\text{fix}(f^2, x, N_2))v_2\) where \(\text{fix}(f^1, x, N_1) \subseteq \text{fix}(f^2, x, N_2)\) and \(v_1 \subseteq v_2\). By using Lemma 16, we obtain the following as expected.

\[\text{Case } c v_1 \rightarrow ev(c, v_1). \text{ From the inversion of } N_1 \subseteq N_2, \text{ it must be that } N_2 \equiv c v_2 \text{ for some } v_1 \subseteq v_2 \text{ holds. Since } \vdash c v_2 : \tau' \text{ holds, } ev(c, v_2) \text{ is also defined. Using the property of } ev(\cdot, \cdot), \text{ we obtain } ev(c, v_1) \subseteq ev(c, v_2) \text{ as expected.}\]

Case (let \(x^1 = v_1 \text{ in } N_3\)) \rightarrow [v_1/x]N_3. From the inversion of \(N_1 \subseteq N_2\), it must be that \(N_2 \equiv \text{let } x^1 = v_2 \text{ in } N_4\) for some \(\tau_2, v_2\) and \(N_4\) where \(v_1 \subseteq v_2\) and \(x : \tau_1; \text{true } \vdash N_3 \subseteq N_4\) holds. Also, we obtain \(\vdash v_1 : \tau_3\) and \(x : \tau_3; \text{true } \vdash N_3 \subseteq N_4\) for some \(\tau_3\) from the inversion of \(\vdash (\text{let } x^1 = v \text{ in } N_3) : \tau.\) Using Lemma 16, \([v_1/x]N_3 \subseteq [v_2/x]N_4\) holds as expected.

Case if true then \(N_3\) else \(N_4\) \rightarrow \(N_3\). From the inversion of \(N_1 \subseteq N_2\), it must be that \(N_2 \equiv \text{if true then } N_5 \text{ else } N_6\) for some \(N_5\) and \(N_6\) where \(N_3 \subseteq N_5\) and \(N_4 \subseteq N_6\) holds.

Case if false then \(N_3\) else \(N_4\) \rightarrow \(N_3\) is similar to the previous case.

Case let \(x^1 = N_3 \text{ in } N_4\) \rightarrow let \(x^1 = N'_3 \text{ in } N_4\) where \(N_3 \rightarrow N'_3\). From the inversion of \(N_1 \subseteq N_2\), it must be that \(N_2 \equiv (\text{let } x^2 = N_5 \text{ in } N_6)\) where \(N_3 \subseteq N_5\) and \(x : \tau_1; \phi \vdash N_4 \subseteq N_6\). From the induction hypothesis, \(N_5 \rightarrow N'_5\) and \(N'_2 \subseteq N'_5\) holds for some \(N'_5\). Therefore, let \(x^2 = N_5 \text{ in } N_6\) \rightarrow let \(x^2 = N'_5 \text{ in } N_6\) and let \(x^1 = N'_3 \text{ in } N_4\) \subseteq let \(x^2 = N'_5 \text{ in } N_6\) holds as expected.

Other cases are trivial from the induction hypothesis.

**Lemma 18.** (Proposition 3 in paper) Suppose \(\vdash N_1 : \tau\) and \(\vdash N_2 : \tau'\) for some \(\tau\) and \(\tau'\). Then, \(N_1 \subseteq N_2\) and \(N_2 \rightarrow N'_2\) imply one of the following.

- \(N_1 \rightarrow N'_1\) and \(N'_1 \subseteq N'_2\)
- \(N_1 \rightarrow \text{error}.)

**Proof.** By induction on \(N_2 \rightarrow N'_2\).

Case assert(true); \(N_4\) \rightarrow \(N_4\). From the inversion of \(N_1 \subseteq N_2\), it must be that \(N_1 \equiv (\text{assert}(\phi); N_3)\) for some \(\phi\) and \(N_3\) where \(\phi \Rightarrow \text{true}\) and \(N_3 \subseteq N_4\). Since \(\phi\) is a closed predicate, \(\phi \equiv \text{true}\) or \(\phi \equiv \text{false}\) holds.

If \(\phi \equiv \text{true}\), then \(\text{assert}(\phi); N_3 \rightarrow N_3\) and \(N_3 \subseteq N_4\) holds as expected. Otherwise, \(\text{assert}(\phi); N_3 \rightarrow \text{error}\) as expected.

Case (\(\lambda x^2.N_4\)) \(v_2 \rightarrow [v_2/x]N_4\). From the inversion of \(N_1 \subseteq N_2\), it must be that \(N_1 \equiv (\lambda x^2.N_3.v_1) \rightarrow [v_1/x]N_3\) for some \(\tau_1, v_1\) and \(N_3\) where \(\tau_1 \subseteq \tau_2\), \(v_1 \subseteq v_2\) and \(x : \tau_1; \text{true } \vdash N_3 \subseteq N_4\). By inverting \(\vdash N_1 : \tau\), we obtain \(\vdash v_1 : \tau_1\). Therefore, using Lemma 16, \([v_1/x]N_3 \subseteq [v_2/x]N_1\) holds as expected.

Case (fix(\(f^2, x, N_4\)) \(v_2 \rightarrow [v_2/x, \text{fix}(f^2, x, N_4)/f]\) \(N_4\) is immediate using Lemma 16.
– Case $cv_2 \rightarrow ev(c, v_2)$. From the inversion of $N_1 \sqsubseteq N_2$, it must be that $N_1 \equiv cv_1$ for some $v_1$ where $v_1 \sqsubseteq v_2$ holds. Since $\vdash cv_1 : \tau$ holds, $ev(c, v_1)$ is also defined. Using the property of $ev(\cdot, \cdot)$, we obtain $ev(c, v_1) \sqsubseteq ev(c, v_2)$ as expected.

– Case $\text{let } x^{\tau_2} = v_2 \in N_4 \rightarrow [v_2/x]N_4$. From the inversion of $N_1 \sqsubseteq N_2$, it must be that $N_1 \equiv \text{let } x^{\tau_1} = v_1 \in N_3$ for some $\tau_1, v_1$ and $N_3$ where $\tau_1 \sqsubseteq \tau_2$, $v_1 \sqsubseteq v_2$ and $x : \tau_1; \text{true} \vdash N_3 \subseteq N_4$.

  Also, $\vdash v_1 : \tau_3$ and $x : \tau_3; \text{true} \vdash N_3 \sqsubseteq N_4$ holds for some $\tau_3$ from the inversion of $\vdash N_1 : \tau$. Therefore, from Lemma 16, we obtain $[v_1/x]N_3 \sqsubseteq [v_2/x]N_4$ as expected.

– Other cases are trivial.

**Lemma 19.** $\Gamma \sqsubseteq \Gamma'$, $\text{dom} (\Gamma) \vdash M \sqsubseteq M'$ and $\Gamma ; \varphi \vdash M \rightsquigarrow N : \tau$ imply $\Gamma' ; \varphi \vdash M' \rightsquigarrow N' : \tau'$, $\Gamma ; \varphi \vdash N \sqsubseteq N'$ and $\text{dom} (\Gamma) \vdash \tau \sqsubseteq \tau'$ for some $N'$ and $\tau'$.

**Proof.** By induction on the derivation of $\Gamma ; \varphi \vdash M \sqsubseteq M'$.

– Case (PM-LAM).

\[
\frac{\text{dom} (\Gamma) \vdash \tau_1 \sqsubseteq \tau'_1 \quad \text{dom} (\Gamma), x \vdash M_1 \sqsubseteq M'_1}{\text{dom} (\Gamma) \vdash \lambda x : \tau_1. M_1 \sqsubseteq \lambda x : \tau'_1. M'_1}
\]

The derivation of $\Gamma ; \varphi \vdash \lambda x : \tau_1. M_1 \rightsquigarrow N_1 : \tau_2$ must be of the following form.

\[
\text{dom} (\Gamma) \vdash \varphi \vdash \lambda x : \tau_1. M_1 \rightsquigarrow \lambda x : \tau_1. N_1 : x : \tau_1 \to \tau_2
\]

Let $\Gamma'$ such that $\Gamma \sqsubseteq \Gamma'$. Since $\Gamma, x : \tau_1 \sqsubseteq \Gamma', x : \tau'_1$ holds, we can apply the induction hypothesis to obtain $\Gamma', x : \tau'_1; \varphi \vdash M'_1 \rightsquigarrow N'_1 : \tau'_2$, $\Gamma, x : \tau_1; \varphi \vdash N_1 \sqsubseteq N'_1$ and $\text{dom} (\Gamma), x \vdash \tau_2 \sqsubseteq \tau'_2$ for some $\tau'_1$ and $\tau'_2$. Therefore, $\Gamma', \varphi \vdash \lambda x : \tau'_1. M'_1 \rightsquigarrow \lambda x : \tau'_1. N'_1 : x : \tau'_1 \to \tau'_2$ holds as expected. We conclude by noting $\text{dom} (\Gamma) \vdash x : \tau_1 \to \tau_2 \sqsubseteq x : \tau'_1 \to \tau'_2$.

– Case (PM-APP).

\[
\frac{\text{dom} (\Gamma) \vdash M_1 \sqsubseteq M'_1}{\text{dom} (\Gamma) \vdash M_1 \, x \sqsubseteq M'_1 \, x}
\]

The derivation of $\Gamma ; \varphi \vdash M_1 \, x \rightsquigarrow N : \tau$ must be of the following form.

\[
\Gamma ; \varphi \vdash M_1 \rightsquigarrow N_1 : y : \tau_1 \to \tau_2 \quad \Gamma (x) = \tau_3 \quad \Gamma ; \varphi \vdash \tau_3 \sqsubseteq \tau_1 \rightsquigarrow N_2
\]

\[
\Gamma ; \varphi \vdash M_1 \, x \rightsquigarrow (\text{let } y^{\tau_1} = N_2 \, x \, \text{in } y) : [x/y]\tau_2
\]

Let $\Gamma'$ such that $\Gamma \sqsubseteq \Gamma'$. From the induction hypothesis, $\Gamma' ; \varphi \vdash M'_1 \rightsquigarrow N'_1 : y : \tau'_1 \to \tau'_2$, $\Gamma ; \varphi \vdash N_1 \sqsubseteq N'_1$ and $\text{dom} (\Gamma) \vdash y : \tau_1 \to \tau_2 \sqsubseteq \tau'$ holds for some $\tau'_1$ and $\tau'$. By inverting $\text{dom} (\Gamma) \vdash y : \tau_1 \to \tau_2 \sqsubseteq \tau'$, it must be that $\tau' \equiv y : \tau'_1 \to \tau'_2$ for some $\tau'_1$ and $\tau'_2$ where $\text{dom} (\Gamma) \vdash \tau_1 \sqsubseteq \tau'_1$ and $\text{dom} (\Gamma), y \vdash \tau_2 \sqsubseteq \tau'_2$. Also, $\Gamma'(x) = \tau'_3$ and $\text{dom} (\Gamma) \vdash \tau'_3 \sqsubseteq \tau_3$ holds for some $\tau'_3$ from $\Gamma \sqsubseteq \Gamma'$. 
From Lemma 3 and Lemma 4, we obtain $\Gamma' ; \varphi \vdash \tau_1' \preceq \tau_1 \rightsquigarrow N_2'$ for some $N_2'$ where $\Gamma ; \varphi \vdash N_2 \subseteq N_2'$ holds. Therefore,

$$\Gamma' ; \varphi \vdash M_1' x \rightsquigarrow (\text{let } y \tau_1' = N_2' x \text{ in } N_1 y) : [x/y] \tau_1'$$

follows. We conclude by noting $\text{dom}(\Gamma') \vdash [x/y] \tau_2 \subseteq [x/y] \tau_2'$ since the underlying types for $\tau_1$ and $\tau_3$ are the same, and the following.

$$\Gamma ; \varphi \vdash (\text{let } y \tau_1' = N_2 x \text{ in } N_1 y) \subseteq (\text{let } y \tau_1' = N_2' x \text{ in } N_1 y)$$

- Case (PM-LET).

\[
\begin{align*}
\text{dom}(\Gamma') &\subseteq M_1' \qquad \text{dom}(\Gamma), x \vdash M_2 \subseteq M_2' \\
\text{dom}(\Gamma) &\vdash (\text{let } x \tau_1 = M_1 \text{ in } M_2) \subseteq (\text{let } x \tau_1 = M_1' \text{ in } M_2')
\end{align*}
\]

The derivation of $\Gamma ; \varphi \vdash \text{let } x \tau_1 = M_1 \text{ in } M_2 \rightsquigarrow N : \tau$ must be the following form.

\[
\begin{align*}
\Gamma ; \varphi \vdash M_1' \rightsquigarrow N_1' &\qquad \Gamma, x : \tau_1 \vdash M_2 \rightsquigarrow N_2' \\
\Gamma, x : \tau_1 \vdash \tau_2 \preceq \tau &\rightsquigarrow N_3 & \text{BT}(\Gamma) \vdash \tau
\end{align*}
\]

$$\Gamma ; \varphi \vdash (\text{let } x \tau_1 = M_1 \text{ in } M_2) \rightsquigarrow (\text{let } x \tau_1 = N_1 \text{ in } \text{let } y \tau_2 = N_2 \text{ in } N_3 y) : \tau$$

Let $\Gamma'$ such that $\Gamma \subseteq \Gamma'$. From the induction hypothesis, $\Gamma' \vdash M_1' \rightsquigarrow N_1' : \tau_1'$ and $\Gamma' \vdash M_2' \rightsquigarrow N_2' : \tau_2'$ for some $N_1'$, $N_2'$, $\tau_1'$ and $\tau_2'$ where $\Gamma ; \varphi \vdash N_1 \subseteq N_1'$, $\Gamma, x : \tau_1 ; \varphi \vdash N_2 \subseteq N_2'$, dom($\Gamma$) $\vdash \tau_1 \subseteq \tau_1'$ and dom($\Gamma$), $x \vdash \tau_2 \subseteq \tau_2'$. From Lemma 13 and Lemma 4, we obtain $\Gamma', x : \tau_1 ; \varphi ; \tau_2' \preceq \tau \rightsquigarrow N_3'$ for some $N_3'$ where $\Gamma, x : \tau_1 ; \varphi \vdash N_3 \subseteq N_3'$ holds, since $\Gamma, x : \tau_1 \subseteq \Gamma', x : \tau_1'$.

Therefore,

$$\Gamma ; \varphi \vdash (\text{let } x \tau_1 = M_1' \text{ in } M_2') \rightsquigarrow (\text{let } x \tau_1 = N_1' \text{ in } \text{let } y \tau_2 = N_2' \text{ in } N_3 y) : \tau$$

holds as expected. We conclude by noting

$$\Gamma ; \varphi \vdash (\text{let } x \tau_1 = N_1 \text{ in } \text{let } y \tau_2 = N_2 \text{ in } N_3 y) \subseteq (\text{let } x \tau_1 = N_1' \text{ in } \text{let } y \tau_2 = N_2' \text{ in } N_3 y)$$

- Case (PM-FIX).

\[
\begin{align*}
\text{dom}(\Gamma'), f, x \vdash M_1' &\subseteq M_1' \\
\text{dom}(\Gamma) &\vdash \text{fix}(f : (x : \tau_1 \rightarrow \tau_2), x, M_1) \subseteq \text{fix}(f : (x : \tau_1' \rightarrow \tau_2'), x, M_1')
\end{align*}
\]

The derivation of $\Gamma ; \varphi \vdash \text{fix}(f : (x : \tau_1 \rightarrow \tau_2), x, M_1) \rightsquigarrow N : \tau$ must be of the following form. (Note $\tau \equiv x : \tau_1 \rightarrow \tau_2$)

\[
\begin{align*}
\Gamma_1 = \Gamma, f : (x : \tau_1 \rightarrow \tau_2), x : \tau_1 &\qquad \Gamma_1 ; \varphi \vdash M_1 \rightsquigarrow N_1 : \tau_3 \\
\Gamma_1 ; \varphi ; \tau_3 \preceq \tau_2 &\rightsquigarrow N_2
\end{align*}
\]

$$\Gamma ; \varphi \vdash \text{fix}(f : (x : \tau_1 \rightarrow \tau_2), x, M_1) \rightsquigarrow \text{fix}(f : (x : \tau_1 \rightarrow \tau_2), x, \text{let } y \tau_3 = N_1 \text{ in } N_2 y) : x : \tau_1 \rightarrow \tau_2$$

Let $\Gamma'$ such that $\Gamma \subseteq \Gamma'$, and $\Gamma'_1 = \Gamma, f : (x : \tau_1' \rightarrow \tau_2'), x : \tau_1'$. Then, $\Gamma_1 \subseteq \Gamma'_1$ holds. From the induction hypothesis, $\Gamma'_1 ; \varphi \vdash M_1 \rightsquigarrow N_1 : \tau_3$ holds for some $N_1$ and $\tau_3$ where $\Gamma_1 ; \varphi \vdash N_1 \subseteq N_1'$ and dom($\Gamma$), $f \vdash \tau_3 \subseteq \tau_3'$. Also, dom($\Gamma$), $f \vdash \tau_2 \subseteq \tau_2'$ holds from the assumption of (PM-Fix). Using
Lemma 13 and Lemma 4, $Γ'; ϕ \vdash τ'_1 \leq τ'_2 \leadsto N'_2 \proves$ for some $N'_2$ where $Γ; ϕ \vdash N_2 \subseteq N'_2$ holds. Therefore,

$Γ; ϕ \vdash \text{fix}(f : x; τ'_1 \rightarrow τ'_2), x, M') \leadsto \text{fix}(f^{x; τ'_1 \rightarrow τ'_2}, x, \text{let } y; τ'_3 = N'_1 \text{ in } N'_2 y) : x; τ'_1 \rightarrow τ'_2$

holds as expected. We conclude by noting $\text{dom}(Γ) \vdash x; τ_1 \rightarrow τ_2 \subseteq x; τ'_1 \rightarrow τ'_2$

and

$Γ; ϕ \vdash \text{fix}(f^{x; τ_1 \rightarrow τ_2}, x, \text{let } y; τ_3 = N_1 \text{ in } N_2 y) \subseteq \text{fix}(f^{x; τ'_1 \rightarrow τ'_2}, x, \text{let } y; τ'_3 = N'_1 \text{ in } N'_2 y)$

– Case (PM-Ir).

$$\text{dom}(Γ) \vdash M_1 \subseteq M'_1 \quad \text{dom}(Γ) \vdash M_2 \subseteq M'_2$$

$$\text{dom}(Γ) \vdash (\text{if } x \text{ then } M_1 \text{ else } M_2) \subseteq (\text{if } x \text{ then } M'_1 \text{ else } M'_2)$$

The derivation of $Γ; ϕ \vdash \text{if } x \text{ then } M_1 \text{ else } M_2 : τ$ must be of the following form.

$$Γ; ϕ \vdash x : \{ v : \text{bool} \mid ϕ' \}
Γ; ϕ \land x \vdash M_1 \leadsto N_1 : τ_1 \quad Γ; ϕ \land x \vdash τ_1 \leq τ \leadsto N_3
Γ; ϕ \land \neg x \vdash M_2 \leadsto N_2 : τ_2 \quad Γ; ϕ \land \neg x \vdash τ_2 \leq τ \leadsto N_4$$

$$Γ; ϕ \vdash \text{if } x \text{ then } M_1 \text{ else } M_2
\leadsto \text{if } x \text{ then } (\text{let } y; τ_1 = N_1 \text{ in } N_3 y) \text{ else } (\text{let } y; τ_2 = N_2 \text{ in } N_4 y) : τ$$

Let $Γ'$ such that $Γ \subseteq Γ'$. We obtain the following from the induction hypothesis.

$$Γ'; ϕ \land x \vdash M'_1 \leadsto N'_1 : τ'_1 \quad Γ; ϕ \land x \vdash N_1 \subseteq N'_1
Γ'; ϕ \land x \vdash M'_2 \leadsto N'_2 : τ'_2 \quad Γ; ϕ \land x \vdash N_2 \subseteq N'_2
\text{dom}(Γ') \vdash τ_1 \subseteq τ'_1 \quad \text{dom}(Γ') \vdash τ_2 \subseteq τ'_2$$

Using Lemma 13, $Γ'; ϕ \land x \vdash τ'_1 \leq τ \leadsto N'_3$ and $Γ'; ϕ \land \neg x \vdash τ'_2 \leq τ \leadsto N'_4$ holds for some $N'_3$ and $N'_4$. From Lemma 4, $Γ; ϕ \land x \vdash N_3 \subseteq N'_3$ and $Γ; ϕ \land \neg x \vdash N_4 \subseteq N'_4$ holds. Therefore, we obtain

$$Γ'; ϕ \vdash \text{if } x \text{ then } M'_1 \text{ else } M'_2
\leadsto \text{if } x \text{ then } (\text{let } y; τ'_1 = N'_1 \text{ in } N'_3 y) \text{ else } (\text{let } y; τ'_2 = N'_2 \text{ in } N'_4 y) : τ$$

as expected. We conclude by noting

$$Γ; ϕ \vdash \text{if } x \text{ then } (\text{let } y; τ_1 = N_1 \text{ in } N_3 y) \text{ else } (\text{let } y; τ_2 = N_2 \text{ in } N_4 y)
\subseteq \text{if } x \text{ then } (\text{let } y; τ'_1 = N'_1 \text{ in } N'_3 y) \text{ else } (\text{let } y; τ'_2 = N'_2 \text{ in } N'_4 y)$$

– Case (PM-ANNOT).

$$\text{dom}(Γ) \vdash M_1 \subseteq M'_1 \quad \text{dom}(Γ) \vdash τ \subseteq τ'
\text{dom}(Γ) \vdash (M_1 : τ) \subseteq (M'_1 : τ')$$

The derivation of $Γ; ϕ \vdash (M_1 : τ) \leadsto N : τ$ must be of the following form.

$$Γ; ϕ \vdash M_1 \leadsto N_1 : τ_1 \quad Γ; ϕ \vdash τ_1 \leq τ \leadsto N_2
Γ; ϕ \vdash (M_1 : τ) \leadsto (\text{let } x; τ_1 = N_1 \text{ in } N_2 x) : τ$$
Let $\Gamma'$ such that $\Gamma \subseteq \Gamma'$. From the induction hypothesis, $\Gamma'; \varphi \vdash M'_1 \leadsto N'_1 \vdash \tau'_1$ for some $N'_1$ and $\tau'_1$ where $dom(\Gamma) \vdash \tau_1 \sqsubseteq \tau'_1$ and $\Gamma; \varphi \vdash N_1 \sqsubseteq N'_1$. Using Lemma 13 and Lemma 4, we have $\Gamma'; \varphi \vdash \tau'_1 \preccurlyeq \tau'' \leadsto N'_2$ for some $N'_2$ where $\Gamma; \varphi \vdash N_2 \sqsubseteq N'_2$ holds. Therefore, we obtain $\Gamma'; \varphi \vdash (M'_1 : \tau') \leadsto (\text{let } x^{\tau_1} = N'_1 \text{ in } N'_2 x) : \tau''$ as expected. We conclude by noting $\Gamma; \varphi \vdash (\text{let } x^{\tau_1} = N_1 \text{ in } N_2 x) \subseteq (\text{let } x^{\tau'_1} = N'_1 \text{ in } N'_2 x)$. 