**Title**
Theory and Applications for Gradual Type Migration



**Author**
Mahmoud, Zeina

**Publication Date**
2023

Peer reviewed|Thesis/dissertation

UNIVERSITY OF CALIFORNIA

Los Angeles

# Theory and Applications for Gradual Type Migration

A dissertation submitted in partial satisfaction

of the requirements for the degree

Doctor of Philosophy in Computer Science

by

Zeina Mohamed Magdy Abdelmigeed Mahmoud

2023

ABSTRACT OF THE DISSERTATION

Theory and Applications for Gradual Type Migration

by

Zeina Mohamed Magdy Abdelmigeed Mahmoud
Doctor of Philosophy in Computer Science
University of California, Los Angeles, 2023
Professor Jens Palsberg, Chair

Gradual typing enables migrating untyped code to typed code by supporting programs with partial type annotations. Supporting programs with partial type annotations enables developers to incrementally add type information to their untyped programs. However, manually annotating code has proven to be an error prone and time consuming task for developers. For this reason, researchers have set out to develop ideas and to construct tools for automatic code annotation with types, that is, a variant of type inference. The particular problem formulations vary based on the specific type system at hand. The solutions range from too theoretical to be implemented in practice to too practical to have a fundamental underlying methodology that allows the solution to be widely adaptable. Because the problem formulations and the proposed solutions vary, it is hard to unify the approaches and apply them to a variety of settings.

We aim to bring order to the space of tool support for gradual types. We focus on automatic code annotation as the primary goal. We accomplish our goals in three steps. First,

we show how to design fundamental algorithms to support automatic code annotation in a gradually typed language. Second, we show that it is possible to design novel gradually typed systems that enable tool support. In particular, we design a Rank-2 intersection type system and a Tensor Type system. Third, we adapt our tool design to tackle the static analysis problem of eliminating branches that depend on shape information, which is a problem we encountered in various popular machine learning benchmarks. The tool is successful on a suite of popular PyTorch benchmarks.

The dissertation of Zeina Mohamed Magdy Abdelmigeed Mahmoud is approved.

Miryung Kim

Todd D. Millstein

Jeremy Siek

Jens Palsberg, Committee Chair

University of California, Los Angeles

2023

*To Matthias Felleisen, who shared the beauty of what we do so effectively that I never stopped, and who - in all my work and years - never stopped sharing his wisdom, encouragement, support, and good nature.*

TABLE OF CONTENTS

LIST OF FIGURES

# ACKNOWLEDGMENTS

None of the work in this dissertation would have been possible without the generous guidance and oversight of my advisor, Jens Palsberg. His kindness and unfailing patience made everything herein more interesting and pleasant to discover. It was also Jens that brought my dear friends, Akshay Utture and Micky Abir, to a lab that was smarter, warmer, and more welcoming for my work because of their presence.

My time at UCLA has been incredibly rewarding on multiple fronts. I've had the privilege of engaging in thought-provoking discussions with my former colleagues, Christian Kalhauge, John Bender and Tianyi Zhang. Christian somehow managed to talk me into using Haskell, which I had never used at the time, to develop a tool for my paper. It worked like a charm!

UCLA gave me Ana Brendel, Noor Nakhaei, and Poorva Garg, who filled work and life with joy. It also gave me Stéphane Pouget who is owed particular thanks for staying up late hours to help me when he did not need to, in keeping with the spirit of a place that I will miss a great deal.

Finally, I give profound thanks to my committee: Miryung Kim encouraged me throughout my journey in computer science, and I admire and aspire to her work ethic. Todd Millstein showed up for me in the toughest parts of that journey and always found time when I needed it. Jeremy Siek pioneered my field, worked with me as an undergraduate research assistant before I came to UCLA, and solidified my interest in doing this work.

# VITA

| | | |
|---|---|---|
| 2017 | UCLA Dean's Graduate Student Research (GSR) Fellowship. | 2021 |
| 2022 | Intern, PyTorch Compiler, Meta, Menlo Park, California. | |

# PUBLICATIONS

**Zeina Migeed** and Jens Palsberg. "What is Decidable about Gradual Types?" In Proceedings of the ACM on Programming Languages, Volume 4, Issue POPL, Article No.: 29pp 1–29. 2020

# CHAPTER 1

# Introduction

## 1.1 Landscape

Programming languages today fall into one of three categories. The first consists of statically typed languages, which require static type annotations for the whole program. The type annotations could be explicit. In other words, they appear in the program text and are provided by the developer. They could also be implicit, allowing the developers to skip them and letting a variant of the Hindley-Milner algorithm restore them [Hin82]. One of the advantages of working with a statically typed language is better code documentation, readability and maintainability. IDEs can exploit types to provide better editing support, such as code completion. Compilers can also utilize type information for code optimization. If the type system is sound with respect to the semantics of the language, then a typed program will satisfy certain guarantees. Statically typed languages only accept programs that adhere to the type system. Therefore, even though a program could have run correctly if we were to ignore the type annotations, the program will not compile anyway if it does not adhere to the system. Examples of statically typed languages include C++ and Java.

The second category consists of dynamically typed languages, which typically require no type annotations from the developer. Programs in this category do not have to satisfy specific type requirements in order to run. Such languages lack the runtime guarantees that

are present in some statically typed languages, however; dynamically typed languages are expressive and flexible. This is due to two reasons. First, they enable the developer to write programs that are almost impossible to assign types to. Second, many of them offer a degree of type safety because they tag values in the program with their appropriate types. These tags make it possible to catch and generate runtime type errors. Examples of dynamically typed languages include PHP, JavaScript and Python.

The third category consists of gradually typed languages. Gradually typed languages can compile programs with type annotations but they do not require them. So gradually typed languages aim to leverage benefits from statically typed and dynamically typed languages. In this dissertation, we use the term "gradual typing" loosely to encompass all languages which allow mixing typed and untyped code. So for this presentation, this category overlaps with dynamically typed languages with type hints.

In addition to allowing mixed type code, gradually typed languages often but not always contain a type which we will refer to as `Dyn` [ST06]. We can view `Dyn` as the type that contains all values. The `Dyn` type enables us to mix between typed and untyped code at a fine granularity, including at the expression or function level. Examples of such languages from industry include TypeScript [BAT14], which is a gradually typed version of JavaScript and Hack, which is a gradually typed version of PHP.

Typescript and Hack promise many advantages compared to JavaScript and PHP. One of their advantages that they support type annotations, which IDEs can leverage. They also support type consistency checking as well as integration with JavaScript and PHP, which makes them as flexible and expressive as their untyped siblings. TypeScript and Hack then erase types before the program is run. Therefore, they do not impose significant performance overhead compared to their corresponding untyped languages, which makes them practical.

However, there are gradually typed languages which aim to enforce that the untyped portion of the code does not violate the invariant implied by the type annotations in the surface syntax. Examples of such langauges include Typed Racket [TFF17] and Reticulated Python [VSS17]. Many researches found that such checks can impose significant runtime overhead [TFG16, GM18]. Felleisen and Greenman [GF18] formalized degrees of runtime type enforcement in gradually typed programs and how they relate to performance overhead in gradually typed programs.

A popular use case for gradual types is to incrementally make untyped programs typed. Untyped programs are written in dynamically typed languages. Thus, the first step of adding types to untyped programs is to consider a dynamically typed language and add gradual typing support to it. The second step is to annotate all programs with `Dyn`. Untyped programs can then be viewed as gradually typed programs with `Dyn` annotations. Finally, we can now incrementally replace some of the `Dyn` annotations in our programs with more precise types. A common goal is to want to replace all `Dyn` occurrences with different types. However, this is not always possible. The reason is that a gradually typed system can be viewed as a super-set of a static and dynamic systems, which allow us to type-check partially typed programs. But the static system alone could sometimes be too restrictive to allow expressing certain programs, which is why we may not always be able to remove all `Dyn` occurrences in a gradually typed program. Another consequence of annotating gradually typed programs with more precise types is that the programs may fail with a type error. The reason this could happen is that when we replace a `Dyn` occurrence with more precise types, we may pick the wrong replacement type, which can introduce a static type error which hasn't previously existed for the program.

## 1.2 Thesis statement

The motivation for this dissertation is the third step of taking untyped code to typed code, which is replacing `Dyn` annotations with more precise types. The task of manually annotating programs requires a lot of effort from the developer because it is error prone [HFS22, ST13, KM17, WMW17, FM14, CT21]. This makes the task quite tedious and time consuming. We observe that in order to realize the full vision of gradual types, we must add a level of automation to annotating gradually typed programs. This observation led us to two paths. First, we can answer fundamental tool support questions. However, the more complex a type system is, the harder it can be to answer tool support questions for it. Second, we may design gradually typed languages that facilitate tool support. However, we must simultaneously account for the gradual typing criteria for our language design and those criteria may clash with our goal for automatic tool support, asking us to make difficult choices.

To address the two paths mentioned above, we provide a methodology for representing code annotation problems in gradual types via a constraint based approach. Next, we design languages that enable automatic code annotations. We observe that a syntactic interpretation of gradual types simplifies our analysis and employ it in our design. We observe that it can be tricky to satisfy all gradual typing criteria while simultaneously providing tool support for gradually typed languages. We also observe that our core methodology for providing tool support for gradually typed languages with a constraint based approach is adaptable to different type systems and different problem formalizations. These observations led us to the following thesis statement.

*The design of gradually typed languages can and should take into account tool support for adding and exploiting types.*

Equipping some gradually typed languages with tool support for automatic code annotation is possible. However, the more complex and expressive a language is, the harder it can be to provide such tool support. With that in mind, we can design gradually typed languages that take tool support into consideration. This imposes restrictions on the language design which may hinder its ability to preserve certain gradual dynamic guarantees. Moreover, our core methodology for addressing problems related to automatic code annotations is adaptable to various type systems and different problem specifications.

## 1.3 Dissertation Overview

In Chapter 2 we give detailed background on gradual typing and consider different gradual typing designs. We then delve into our specific landscape, which is automatic code annotation on gradual types. We go over variants of the problem, as well as related problems from the literature.

In Chapter 3, we present the first step we take to address the problem of tool support in gradual types, where we consider the simplest language and the simplest type system possible, namely, the Gradually Typed Lambda Calculus (GTLC) [CS16]. We ask fundamental questions that can help us automatic code annotation in this setting. The questions are:

1. Can a program be typed any further?
2. Is there a version of the program that is more precise than all other versions?
3. Are there finitely many ways to annotate the program?
4. Is there a way to annotate the program such that it cannot be typed any further?

We discover that the complexities of these four questions range from polynomial to NP-hard, leading us to believe that automatic code annotation for gradually typed languages is hard. When considering a more complex type system, we observe that answering the above three questions does not get easier, which leads us to our next observation.

In chapter 4, we design a Rank-2 intersection type system that can support gradual types, as well as automatic code annotation for them. The reason we consider a Rank-2 intersection type system, as opposed to a full-fledged intersection type system, is that type inference for Rank-2 intersections is decidable [Jim95], but it is undecidable for unrestricted ranks [CLP19]. We set out to satisfy gradual typing guarantees, while providing code annotation tool support and observe that it is difficult to satisfy all criteria at once. For this type system, we could not satisfy the gradual typing dynamic criteria from Siek at al. [SVC15a]. We discuss the challenges and the trade offs when it comes to satisfying code automation criteria as well as gradual typing criteria.

In chapter 5, we aim to scale our ideas for automatic code annotations. With the rise of AI programs, we observe a need for languages that can reason about partial type information and that can support tensors, which is a central data structure in AI programs. Tensors are a useful abstraction which makes writing AI programs easier. It is used in many popular machine learning frameworks such as PyTorch [PGC17] and TensorFlow [AAB16]. Thus, we design a gradually typed language that supports tensor shapes where we view shapes as type annotations for tensors. Our language satisfies static criteria from Siel et al. [SVC15a] but we leave dynamic criteria to future work. Our methodology from the previous two works carried over into our tensor language, but we tweak the set of questions that we ask about tool support in order to suit the developer needs for AI programs. Thus, we ask the following three questions:

1. Can we remove all `Dyn` occurrences from the progoram?

2. Given a set arithmetic constraints, can we find valid type annotations that satisfies them?

3. Can we eliminate branches that depend on the input for all inputs to the program?

In addition to tacking the problem from a theoretical perspective, we build a tool that demonstrates the practicality of our approach on a suite of PyTorch benchmarks.

## 1.4 Contributions

The contributions for this dissertation can be summarized as follows:

1. A formalization of fundamental questions about automatic code annotations and their complexities for the Gradually Typed $\lambda$-calculus.

2. A gradual Rank-2 intersection type system which enables gradual typing tool support

3. A gradual Tensor language which enables tool support and a tool that demonstrates such tool support

4. An adaptation of the solution to automatic code annotation to the branch elimination problem

# CHAPTER 2

# Type Inference, Type Migration and Algorithmic Support

Since type migration is closely related to type inference, we begin with a brief survey of type inference. We then discuss type migration.

## 2.1 An overview of Type Inference

There are several notions of type inference. The first assumes a program with some missing types and aims to restore them. This is known as Hindly Milner type inference [Mil78]. The inference algorithm accepts a program, generates equational constraints and then solves them using unification. That approach is straightforward and works well for simple types [GR13, RY08, Rem05]. However, it fails on richer type systems. For example, type inference with polymorphic recursion [Hen93], intersection types [KW04] or shapes [HKS22] is undecidable. This led researchers to consider decidable restrictions of such systems, such as Rank-2 intersection types [Jim95]. In an attempt to expand the uses of Hindley Milner inference to more settings, soft typing attempts to adapt this inference style to untyped programming languages [CF91], but the next approach is far more successful in this direction.

The next approach of type inference assumes untyped code and uses static analysis to infer types [JMT09, PS91, FFK96]. One approach in this category uses set-based analysis. For example, Palsberg and Schwartzbach [PS91] utilize set-based analysis to infer sets of

value types for object oriented languages. Their inferencer accepts a program, generates subset constraints for it and then computes the transitive closure propagated through set constructors. The result of the transitive-closure computation is a conservative approximation of the sets of values that each expression in the program computes. A set-based approach has been used for various languages but it falls short when it comes to inferring polymorphic types.

The first two approaches only apply to whole modules which makes them less adaptable to practice because it is harder to locate and debug static type errors. To solve this problem, researchers have considered local type inference [PT00]. Local type inference is practical and efficient but the inferred types can be incorrect for whole programs. Thus, this inference style is considered as a more limited form of type inference. Despite its flaws, local type inference is especially useful for programming with generics in languages such as Java and Scala.

In the context of gradually typed languages, we have two categories of type inference: static and dynamic. Static type inference includes the inference tool for Python by [HUE18], and the inference tool for Dart by [HMS16]. Some approaches to type inference add types for the purpose of program understanding but without a guarantee that the resulting program type checks. A recent example is the inference tool for Python [XZC16]. Static type inference also includes the foundational work by Siek et al. [SV08] and Garcia and Cimini [GC15]. One property of the inference algorithm in Garcia and Cimini [GC15] is that it outputs a type containing type variables whose instantiation is not decided. This raises the question of what to do with those type variables. This bring us to dynamic type inference, which includes Miyazaki et al. [MSI18]. The paper by Miyazaki et al. [MSI18] builds on the work of Garcia and Cimini [GC15] but proposes to delay the decision about what to do with the type variable until runtime, when those variables are instantiated.

## 2.2 What is Type Migration?

Type Migration is a way to formalize automatic program annotation in gradual types. It was first defined in terms of the Gradually Typed Lambda Calculus (GTLC) [ST06] but has later been extended to richer languages. Type migration is closely related to type inference but there are several fundamental differences between them.

### 2.2.1 The Gradually Typed Lambda Calculus

Our starting point for studying type migration is the GTLC. It is the gradually typed version of the Simply Typed Lambda Calculus (STLC). The GTLC contains a type `Dyn` which represents the absence of type information. That type gives rise to a binary relation called *type precision* which allows us to compare programs by how static they are. Program $A$ is more precise than Program $B$ if we can obtain program $B$ by replacing some of the types in program $A$ with `Dyn`.

### 2.2.2 The gradual typing criteria

The GTLC satisfies the static and dynamic properties of Siek et al. [SVC15a]. The properties aim to ensure that the gradual language is safe and that it meets developer expectations statically as well as dynamically. The most notable criteria is the *Gradual Guarantee*, which is divided into static and dynamic. The static gradual guarantee states that when making a well-typed program less precise, the program must continue to type check. All systems that we will consider in this dissertation satisfy the static gradual guarantee. In constrast, the dynamic gradual guarantee handles runtime enforcement of gradual types. In this dissertation,

different languages satisfy different static and dynamic criteria. We will discuss this in detail in upcoming chapters.

### 2.2.3 Type Migration

Type migration is the process of making a program more precise, while maintaining its well-typedness. It is useful for porting gradually typed code from untyped to typed, which is key for leveraging gradual typing benefits.

### 2.2.4 Algorithmic support

Algorithmic support for type migration is important for migrating unannotated legacy code in gradual languages. The task of manually annotating code is tedious and time consuming, which is a barrier to taking full advantage of gradual types. In this dissertation, we focus on whole program type migration. We hope that future work extends automatic program annotation that supports modularity.

Consider a gradually typed program. Generally, when we face a migration task in this setting, the first thing we want to know is whether any improvement is possible; if not, then we are done right away. Otherwise, the best we can hope for is that one of the migrations is the *greatest* in the $\sqsubseteq$-order; we call it the *top* migration. In the absence of a top choice, we can ask whether the set of migrations is finite. If so, then we have a finite set of migrations that cannot be improved in the $\sqsubseteq$-order; they are *maximal*. We must pick one, even though none of them is $\sqsubseteq$-greater than the others. If the migration space is *finite*, then we can find all the maximal migrations by iterating through the migration space. Even, if it is infinite, there may still be maximal migrations, in which case we can choose one of them. Or perhaps the migration space has no maximal element at all.

We distill the properties mentioned above into four key questions to ask of every migration problem:

1. *is any improvement possible?* (Singleton problem)
2. *does a top migration exist?* (Top-choice problem),
3. *is the set of migrations finite?* (Finiteness problem)
4. *does a maximal migration exist?* (Maximality problem)

Information about which kind of program we are facing will help the developer figure out how long we should continue a migration exploration. Figure 2.1 provides a pictorial representation of the problems.

In Chapter 3, we present algorithms (with names in bold below) and a hardness result for deciding the four questions above for the gradually typed $\lambda$-calculus [ST06].

In Chapter 4, we extend the system with gradual intersection types and prove that the Singleton problem, Top-Choice problem and Finiteness problem have the same complexities as those of the GTLC. By contrast, for Chapter 5 which migrates tensor shapes, we present variants of these questions that are better matches to a developer's needs in this context than those listed here.

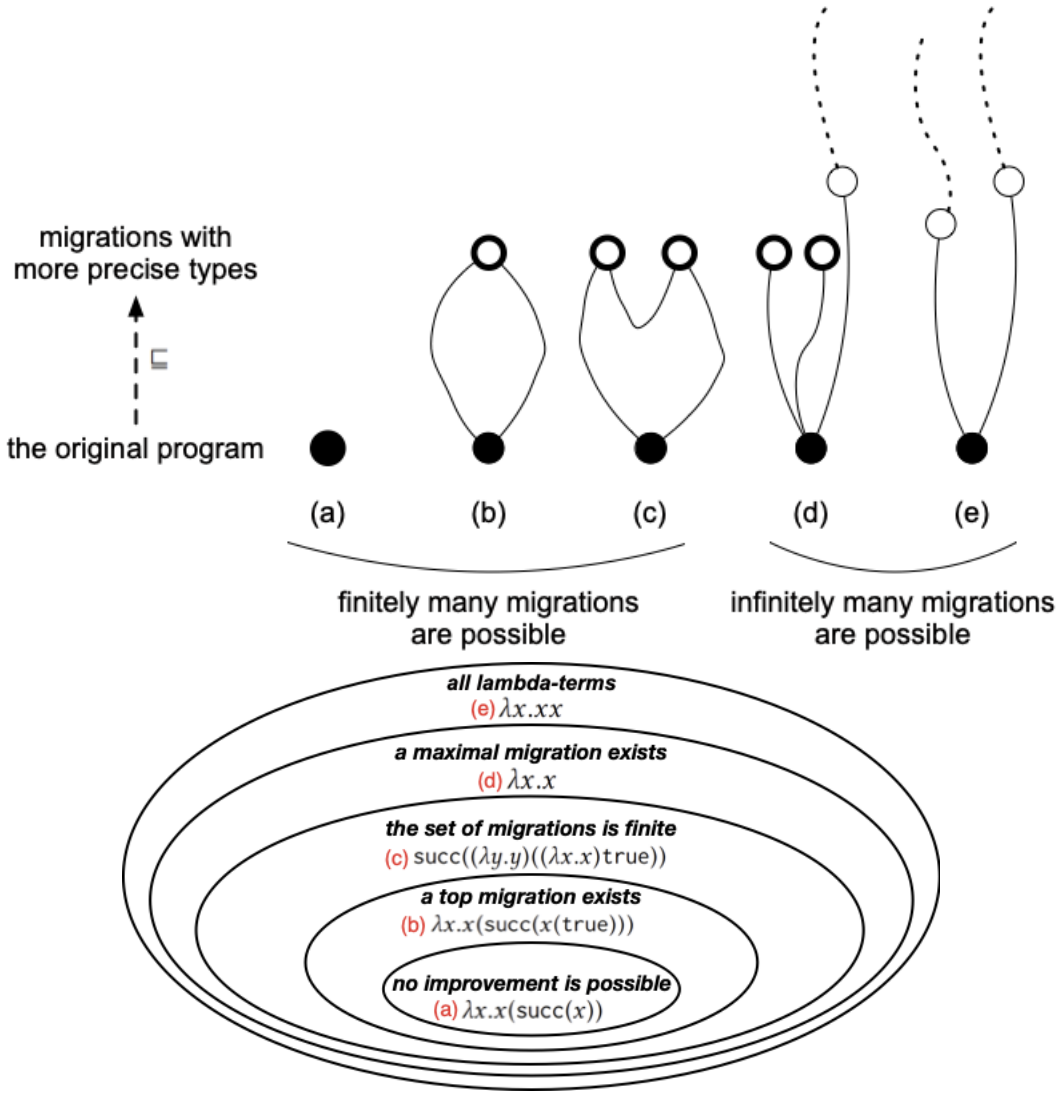


**Figure 2.1:** Nested sets of $\lambda$-terms: (a) no improvement is possible; (b) a top migration exists; (c) the set of migrations is finite; (d) a maximal migration exists; (e) all lambda-terms.

# CHAPTER 3

# Migrating Gradual Simple Types

In this chapter, our goal is to provide a foundation for better tool support by settling decidability questions about migration with gradual types. We present three algorithms and a hardness result for deciding key properties and we explain how they can be useful during an exploration. In particular, we show how to decide whether the migration space is finite, whether it has a top element, and whether it is a singleton. We also show that deciding whether it has a maximal element is NP-hard. Our implementation of our algorithms worked as expected on a suite of microbenchmarks.

## 3.1 Introduction

**Background.** Static type checking has led to more reliable and faster software because types make programs more readable, prevent entire classes of mistakes, and help compilers optimize data layout and data access. By contrast, dynamically typed languages allow programmers to quickly prototype systems and build programs that are correct but fit no particular type system. The complementary strengths of static and dynamic typing have led researchers to explore ways to combine them. In this chapter we will focus on one such combination, namely the well-known *gradual typing* of [ST06]. Gradual typing combines static typing with a dynamic type that we will write as `Dyn`. One way to take advantage of

`Dyn` is to use static types as much as possible and use `Dyn` otherwise. Gradual typing enables programmers to get the discipline of static typing and the freedom of dynamic typing in a way that gives well-understood benefits [SVC15a].

Gradual typing has found practical application in Typed Racket [TF08], TypeScript [BAT14], Reticulated Python [VKS14], and others. In each case a programmer can view a program in the original dynamic language (Racket, JavaScript, and Python) as a program in the gradually typed variant where all types are `Dyn`. Then the goal of *type migration* is to change some of the `Dyn` types to more precise types. This goal was formalized by [ST06] who defined ?a binary precision order $\sqsubseteq$ on types, including $\texttt{Dyn} \sqsubseteq \texttt{int}$ and $(\texttt{Dyn} \to \texttt{Dyn}) \sqsubseteq (\texttt{Dyn} \to \texttt{bool})$; the type on the right of $\sqsubseteq$ is more precise. Similarly, the precision order $\sqsubseteq$ on terms that says that $E \sqsubseteq E'$ if $E'$ has more precise type annotations than $E$. For example, $(\lambda x : \texttt{Dyn}.x) \sqsubseteq (\lambda x. : \texttt{int}.x)$, where we improve `Dyn` to `int`. Thus, the goal of type migration of $E$ is to find $E'$ such that $E \sqsubseteq E'$ and $E'$ type checks in the gradual typing discipline. Ultimately, programmers may prefer to find an $E'$ that is the top element of the migration space, if it exists, or else find a maximal element that cannot be improved.

We consider the four key problems which we discussed in Chapter 2. Specifically, our singleton checker decides whether any improvement is possible, our top-choice checker decides whether a top migration exists, and our finiteness checker decides whether the set of possible migrations is finite. We also show that the maximality problem is NP-hard. Our results can be summarized as follows:

| | | |
|---|---|---|
| Singleton problem: | decidable in $O(n^2)$ time (Theorem 3.9) | **Singleton Checker** |
| Top-choice problem: | decidable in EXPTIME (Theorem 3.23) | **Top-Choice Checker** |
| Finiteness problem: | decidable in EXPTIME (Theorem 3.20) | **Finiteness Checker** |
| Maximality problem: | NP-hard (Theorem 3.24). | |

In Section 3.2 we recall the gradually typed lambda calculus, we formalize the four key properties as decision problems, and we give examples of programs with different properties. Our singleton checker (Section 3.3) relies on a theorem known as *the static gradual guarantee* [SVC15a] and on a type checker. The idea is to try all one-step improvements (that each replaces a single occurrence of `Dyn`) and see if any of them type check. If none of those improvements type checks, then no improvement is possible. Our finiteness checker (Section 3.4) uses type constraints. Specifically, it represents the set of possible migrations as the set of solutions to constraints that it generates from the program. Then, it decides whether the set of solutions is finite. Our top-choice checker (Section 3.5) first runs our finiteness checker and then searches the set of migrations. Our NP-hardness proof (Section 3.6) reduces 3SAT to maximality: it maps a 3SAT formula to a program in such a way that the formula is satisfiable if and only if the program has a maximal migration.

Our implementation of our algorithms worked as expected on a suite of microbenchmarks (Section 3.7). We discuss related work in Section 3.8. Briefly, the most closely related work is the POPL 2018 paper by Campora, Chen, Erwig, and Walkingshaw [CCE18], which presented an efficient approach to migrating a program, but did not address the four problems listed above. We use an entirely different approach, in part because the approach in [CCE18] may produce non-maximal migrations (see Sections 3.7–3.8), which makes it unsuitable for our decision problems.

An extended version of the chapter is available from our website; it has supplementary material that consists of four appendices.

## 3.2 The Gradually Typed Lambda Calculus

### 3.2.1 Syntax and Type System

Figure 3.1 shows the gradually typed $\lambda$-calculus [ST06], in the convenient reformulation by [CS16]. We use $n$ to range over natural numbers and we use $x$ to range over term variables. Types include the special type `Dyn`, as well as two base types `bool` and `int`, and function types $T \to T$. Terms include Booleans, natural numbers, variables, abstractions, and applications. The typing rules for Booleans (*T-True* and *T-False*) and numbers (*T-Num*) are straightforward, and the typing rules for variables (*T-Var*) and abstractions (*T-Abs*) are as in simply-typed $\lambda$-calculus. The type rule for applications (*T-App*) uses notions of matching and consistency to make it more flexible than the rule for applications in simply-typed $\lambda$-calculus. Specifically, the use of matching $T_1 \rhd (T_{11} \to T_{12})$ allows $T_1$ to be `Dyn`, in which case $T_{11}$ and $T_{12}$ are also `Dyn`, as expressed in (*M-Dyn*). Additionally, the use of consistency $T_2 \sim T_{11}$ allows the type $T_2$ to have a relationship with $T_{11}$ that is weaker than equality. Most notably, the rules ($T \sim$ `Dyn`) (*C-Dyn1*) and (`Dyn` $\sim T$) (*C-Dyn2*) define that any type is consistent with `Dyn`. Note that while $\sim$ is reflexive and symmetric, it fails to be transitive, which is an essential part of the design of the entire calculus. The precision relations on types and terms are as we introduced them briefly in Section 5.1.

Next we state four properties that will be useful throughout the chapter.

**Theorem 3.1** (Unique Type)**.** $\forall E, \Gamma, T, T',$ *if* $\Gamma \vdash E : T$ *and* $\Gamma \vdash E : T',$ *then* $T = T'$.

**Theorem 3.2** (Weakening)**.** $\forall E, \Gamma, T, T' :$ *if* $x \notin FV(E),$ *then* $\Gamma \vdash E : T$ *iff* $\Gamma, x : T' \vdash E : T$.

**Theorem 3.3** (Static Gradual Guarantee [SVC15a])**.** $\forall E, E', \Gamma, T : \exists T' :$ *if* $\Gamma \vdash E : T \land E' \sqsubseteq E$ *then* $\Gamma \vdash E' : T' \land T' \sqsubseteq T$.

**Theorem 3.4** (Finite Intervals)**.** $\forall E_l, E_u : \{\ E \mid E_l \sqsubseteq E \sqsubseteq E_u\ \}$ *is finite.*

Syntax:

$$\begin{array}{rrcl} \text{(Types)} & T & ::= & \texttt{Dyn} \mid \texttt{bool} \mid \texttt{int} \mid T \to T \\ \text{(Terms)} & E & ::= & \texttt{true} \mid \texttt{false} \mid n \mid x \mid \lambda x : T.E \mid E\ E \\ \text{(Environments)} & \Gamma & ::= & \emptyset \mid \Gamma, x : T \end{array}$$

Typing rules:

$\Gamma \vdash \texttt{true} : \texttt{bool}$ *(T-True)* $\qquad \Gamma \vdash \texttt{false} : \texttt{bool}$ *(T-False)* $\qquad \Gamma \vdash n : \texttt{int}$ *(T-Num)*

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T}\ \textit{(T-Var)} \qquad \frac{\Gamma, x : T_1 \vdash E : T_2}{\Gamma \vdash (\lambda x : T_1.E) : T_1 \to T_2}\ \textit{(T-Abs)}$$

$$\frac{\begin{array}{cc} \Gamma \vdash E_1 : T_1 & \Gamma \vdash E_2 : T_2 \\ T_1 \rhd (T_{11} \to T_{12}) & T_2 \sim T_{11} \end{array}}{\Gamma \vdash E_1\ E_2 : T_{12}}\ \textit{(T-App)}$$

Consistency:

$T \sim \texttt{Dyn}$ *(C-Dyn1)* $\qquad \texttt{Dyn} \sim T$ *(C-Dyn2)* $\qquad \texttt{bool} \sim \texttt{bool}$ *(C-Bool)*

$$\texttt{int} \sim \texttt{int}\ \textit{(C-Int)} \qquad \frac{T_1 \sim T_3 \quad T_2 \sim T_4}{(T_1 \to T_2) \sim (T_3 \to T_4)}\ \textit{(C-Arrow)}$$

Matching:

$(T_1 \to T_2) \rhd (T_1 \to T_2)$ *(M-Arrow)* $\qquad \texttt{Dyn} \rhd (\texttt{Dyn} \to \texttt{Dyn})$ *(M-Dyn)*

Precision:

$\texttt{Dyn} \sqsubseteq T$ *(P-Dyn)* $\qquad T \sqsubseteq T$ *(P-SameT)* $\qquad \frac{T_1 \sqsubseteq T_3 \quad T_2 \sqsubseteq T_4}{T_1 \to T_2 \sqsubseteq T_3 \to T_4}$ *(P-Arrow)*

$$E \sqsubseteq E\ \textit{(P-SameE)} \qquad \frac{T_1 \sqsubseteq T_2 \quad E_1 \sqsubseteq E_2}{\lambda x : T_1.E_1 \sqsubseteq \lambda x : T_2.E_2}\ \textit{(P-Abs)}$$

$$\frac{E_1 \sqsubseteq E_3 \quad E_2 \sqsubseteq E_4}{(E_1\ E_2) \sqsubseteq (E_3\ E_4)}\ \textit{(P-App)}$$

**Figure 3.1:** The gradually typed $\lambda$-calculus.

The type system assigns at most one type to every program, as expressed by Theorem 3.1. The reason is that every bound variable is declared with a type. Given $E, \Gamma$, we can check in linear time whether $\exists T : \Gamma \vdash E : T$. We have implemented a type checker that carries out this check. Some programs type check, like $(\lambda x : \texttt{int}.\ x)5$, while other programs fail to type check, like $(\lambda x : \texttt{int}.\ x)\texttt{true}$, both programs in context of any environment. Theorem 3.2 is a standard result about adding and removing parts of the environment that is true of many type systems. The static gradual guarantee (Theorem 3.3) says that if an expression type checks and we make the type annotations less precise, then the changed expression also type checks. The precision order guarantees that the set of terms that fit between two terms is finite (Theorem 3.4).

In Appendix A of the supplementary material we prove Theorem 3.1. We omit the standard proof of Theorem 3.2. [SVC15a] proved Theorem 3.3. The proof of Theorem 3.4 is straightforward and omitted.

### 3.2.2 Decision Problems

We define that $E'$ is a $\Gamma$-migration of $E$ (written $E \leq_\Gamma E'$) iff $(E \sqsubseteq E' \ \wedge \ \exists T' : \Gamma \vdash E' : T')$. Intuitively, this means that $E'$ is a $\Gamma$-migration of $E$ if $E'$ improves $E$ and $E'$ type checks.

Given $E$, we define the set of $\Gamma$-migrations of $E$: $Mig_\Gamma(E) = \{E' \mid E \leq_\Gamma E'\}$.

An element $E'$ of $Mig_\Gamma(E)$ is a greatest element if $\forall E'' \in Mig_\Gamma(E) : E'' \sqsubseteq E'$. An element $E'$ of $Mig_\Gamma(E)$ is a maximal element if $\forall E'' \in Mig_\Gamma(E) : (E' \sqsubseteq E'') \Rightarrow (E' = E'')$. In other words, a greatest element is $\sqsubseteq$-greater than all others, while a maximal element cannot be improved. If a greatest element exists, then it is unique, and it is also a maximal element.

For given $E, \Gamma$, our goal is to decide the four questions from Section 5.1 about $Mig_\Gamma(E)$. We formalize those questions as follows:

| | |
|---|---|
| Singleton problem: | is $Mig_\Gamma(E)$ a singleton? |
| Top-choice problem: | does $Mig_\Gamma(E)$ have a greatest element? |
| Finiteness problem: | is $Mig_\Gamma(E)$ finite? |
| Maximality problem: | does $Mig_\Gamma(E)$ have a maximal element? |

### 3.2.3 The Programs in Figure 2.1 Have Different Properties

Figure 2.1 shows five example programs; now we will discuss them in detail.

**No improvement is possible.** Consider $\lambda x.x(\texttt{succ}(x))$. This program uses $x$ as both a function and as an integer. This leaves a single choice for the type of $x$, namely $\texttt{Dyn}$. So, no improvement is possible and $Mig_\Gamma(E)$ is a singleton. In summary:

$$Mig_\Gamma(\ \lambda x : \texttt{Dyn}.\ x(\texttt{succ}(x))\ )\ \ =\ \ \{\ \lambda x : \texttt{Dyn}.\ x(\texttt{succ}(x))\ \}$$

**A top migration exists.** Consider $\lambda x.x(\texttt{succ}(x(\texttt{true})))$. This program applies $x$ to both an integer and to a Boolean. Additionally, the result of applying $x$ is used as an integer. Thus, we have two options for the type of $x$, namely $\texttt{Dyn} \to \texttt{Dyn}$ and $\texttt{Dyn} \to \texttt{int}$. We have that $(\texttt{Dyn} \to \texttt{Dyn}) \sqsubseteq (\texttt{Dyn} \to \texttt{int})$. Thus, for $E = \lambda x.x(\texttt{succ}(x(\texttt{true})))$ and $\Gamma = \texttt{succ} : \{\texttt{int} \to \texttt{int}\}$, we have that $Mig_\Gamma(E)$ has a greatest element, namely the one that annotates $x$ with $\texttt{Dyn} \to \texttt{int}$. In summary,

$$\begin{aligned} Mig_\Gamma(\ \lambda x : \texttt{Dyn}.\ x(\texttt{succ}(x(\texttt{true})))\ )\ \ =\ \ \{\ & \lambda x : \texttt{Dyn}.\ x(\texttt{succ}(x(\texttt{true}))), \\ & \lambda x : (\texttt{Dyn} \to \texttt{Dyn}).\ x(\texttt{succ}(x(\texttt{true}))) \\ & \lambda x : (\texttt{Dyn} \to \texttt{int}).\ x(\texttt{succ}(x(\texttt{true})))\ \} \end{aligned}$$

**Finitely many migrations exist but none is the single best.** Consider $\texttt{succ}((\lambda y.y)((\lambda x.x)\texttt{true}))$. This program binds $x$ to a Boolean, then passes $x$ to $y$, and finally uses $y$ as an integer. This means that for $E = \texttt{succ}((\lambda y.y)((\lambda x.x)\texttt{true}))$ and $\Gamma = \texttt{succ} : \{\texttt{int} \to \texttt{int}\}$, we have that $Mig_\Gamma(E)$ has three elements, namely the ones that annotate $x$ and $y$ as follows: $[x : \texttt{Dyn}; y : \texttt{Dyn}]$ and $[x : \texttt{Dyn}; y : \texttt{int}]$ and $[x : \texttt{bool}; y : \texttt{Dyn}]$. Notice that while $Mig_\Gamma(E)$ is finite, it has no greatest element. In summary,

$$
\begin{aligned}
Mig_\Gamma(\ \texttt{succ}((\lambda y : \texttt{Dyn}.y)((\lambda x : \texttt{Dyn}.x)\texttt{true}))\ ) \quad = \quad \{\ & \texttt{succ}((\lambda y : \texttt{Dyn}.y)((\lambda x : \texttt{Dyn}.x)\texttt{true}))\ ), \\
& \texttt{succ}((\lambda y : \texttt{int}.y)((\lambda x : \texttt{Dyn}.x)\texttt{true}))\ ), \\
& \texttt{succ}((\lambda y : \texttt{Dyn}.y)((\lambda x : \texttt{bool}.x)\texttt{true}))\ ) \\
\}
\end{aligned}
$$

**Infinitely many migrations exist and some are maximal.** Consider $\lambda x.x$. This program has infinitely many migrations, which includes a maximal migration where we give $x$ the type $\texttt{int}$. In summary,

$$Mig_\Gamma(\ \lambda x : \texttt{Dyn}.x\ ) \quad = \quad \{\ \lambda x : \texttt{Dyn}.x\ , \lambda x : \texttt{bool}.x\ , \lambda x : \texttt{int}.x\ , \lambda x : (\texttt{Dyn} \to \texttt{Dyn}).x\ , \ldots\ \}$$

**Every migration can be improved.** Consider $\lambda x.xx$. This program has infinitely many migrations and none of them is maximal. For example, let us give $x$ the type $\texttt{Dyn} \to \texttt{int}$. This makes the program type check because when we apply $x$ to $x$, the type of the argument $x$ (which is $\texttt{Dyn} \to \texttt{int}$) is consistent with the argument type of $x$ (which is $\texttt{Dyn}$). However, we can improve $\texttt{Dyn} \to \texttt{int}$ by giving $x$ the type $(\texttt{Dyn} \to \texttt{Dyn}) \to \texttt{int}$. Notice that $(\texttt{Dyn} \to \texttt{int}) \sqsubseteq ((\texttt{Dyn} \to \texttt{Dyn}) \to \texttt{int})$. Notice also that giving $x$ the type $(\texttt{Dyn} \to \texttt{Dyn}) \to \texttt{int}$ makes the program type check. This is because when we apply $x$ to $x$, the type of the

argument $x$ (which is $((\texttt{Dyn} \to \texttt{Dyn}) \to \texttt{int})$ is consistent with the argument type of $x$ (which is $(\texttt{Dyn} \to \texttt{int})$). In other words, we can check easily that $((\texttt{Dyn} \to \texttt{Dyn}) \to \texttt{int}) \sim (\texttt{Dyn} \to \texttt{Dyn})$. A similar improvement can be made for every type of $x$ that makes the program type check. So, indeed, none of the migrations is maximal. In summary,

$$\begin{aligned} Mig_{\Gamma}(\ \lambda x : \texttt{Dyn}.\ xx\ ) \quad = \quad \{\ & \lambda x : \texttt{Dyn}.\ xx\ , \lambda x : (\texttt{Dyn} \to \texttt{Dyn}).\ xx\ , \quad \lambda x : (\texttt{Dyn} \to \texttt{int}).\ xx\ , \\ & \lambda x : ((\texttt{Dyn} \to \texttt{Dyn}) \to \texttt{int}).\ xx\ , \ldots\ \} \end{aligned}$$

## 3.3 The Singleton Problem

Our algorithm for the singleton problem relies on the static gradual guarantee (Theorem 3.3) and on a type checker for the gradually typed lambda-calculus. The idea is to try all one-step improvements and see if any of them type check. If none of those improvements type checks, then no improvement is possible.

We begin by defining, for a type $T$, the set $\mathcal{S}(T)$ of one-step improvements, and for a term $E$, the set $\mathcal{S}(E)$ of one-step improvements. Intuitively, $\mathcal{S}(T)$ is the set of types that are one step above $T$ in the precision relation. Similarly, $\mathcal{S}(E)$ is the set of terms that are one

step above $E$ in the precision relation. We go one step above by replacing a single occurrence of `Dyn` by either `bool`, `int`, or (`Dyn` $\to$ `Dyn`).

$$
\begin{aligned}
\mathcal{S}(\texttt{bool}) &= \emptyset \\
\mathcal{S}(\texttt{int}) &= \emptyset \\
\mathcal{S}(\texttt{Dyn}) &= \{\ \texttt{bool},\ \texttt{int},\ \texttt{Dyn} \to \texttt{Dyn}\ \} \\
\mathcal{S}(T_1 \to T_2) &= \bigcup_{T_1' \in \mathcal{S}(T_1)} \{\ T_1' \to T_2\ \} \cup \bigcup_{T_2' \in \mathcal{S}(T_2)} \{\ T_1 \to T_2'\ \}
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{S}(n) &= \emptyset \\
\mathcal{S}(\texttt{true}) &= \emptyset \\
\mathcal{S}(\texttt{false}) &= \emptyset \\
\mathcal{S}(x) &= \emptyset \\
\mathcal{S}(\lambda x : T.F) &= \bigcup_{T' \in \mathcal{S}(T)} \{\ \lambda x : T'.F\ \} \cup \bigcup_{F' \in \mathcal{S}(F)} \{\ \lambda x : T.F'\ \} \\
\mathcal{S}(E_1 E_2) &= \bigcup_{E_1' \in \mathcal{S}(E_1)} \{\ E_1' E_2\ \} \cup \bigcup_{E_2' \in \mathcal{S}(E_2)} \{\ E_1 E_2'\ \}
\end{aligned}
$$

For example,

$$
\mathcal{S}(\lambda x : \texttt{Dyn}.x) = \{\ \lambda x : \texttt{bool}.x,\ \ \lambda x : \texttt{int}.x,\ \ \lambda x : (\texttt{Dyn} \to \texttt{Dyn}).x\ \}
$$

Figure 3.2 shows the bottom three levels of the precision order for $\lambda x : \texttt{Dyn}.x$. The idea is to call $\mathcal{S}(\lambda x : \texttt{Dyn}.x)$ to obtain the second "column" of Figure 3.2. Additionally, we can call $\mathcal{S}(\lambda x : \texttt{Dyn} \to \texttt{Dyn}.x)$ to obtain the third "column" of Figure 3.2.

Now we state the correctness of $\mathcal{S}$.

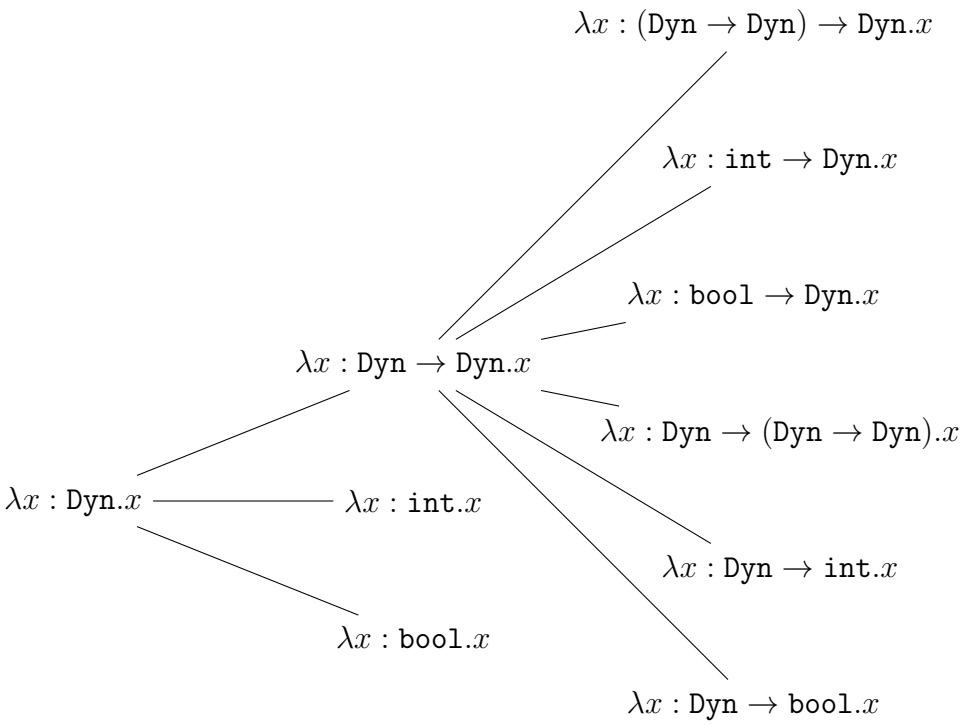


**Figure 3.2:** The bottom three levels of the migration space for $\lambda x : \texttt{Dyn}.x$.

**Theorem 3.5.** $\forall T : \mathcal{S}(T) = \{\ T_u \ \mid\ T_u \neq T \ \wedge\ \forall T' : (T \sqsubseteq T' \sqsubseteq T_u)$ *iff* $((T = T') \vee (T' = T_u))\ \}$.

**Theorem 3.6.** $\forall E : \mathcal{S}(E) = \{\ E_u \ \mid\ E_u \neq E \ \wedge\ \forall E' : (E \sqsubseteq E' \sqsubseteq E_u)$ *iff* $((E = E') \vee (E' = E_u))\ \}$.

The proof of Theorem 3.5 is by straightforward induction on $T$, and the proof of Theorem 3.6 is by straightforward induction on $E$, using Theorem 3.5.

**Theorem 3.7.** $\forall E, \Gamma : Mig_\Gamma(E)$ *is a singleton iff* $\mathcal{S}(E) \cap Mig_\Gamma(E) = \emptyset$.

*Proof.* We will prove the two directions in turn.

**Forwards direction.** Suppose $Mig_\Gamma(E)$ is a singleton, that is $Mig_\Gamma(E) = \{E\}$. We have from Theorem 3.6 that $E \notin \mathcal{S}(E)$ so $\mathcal{S}(E) \cap Mig_\Gamma(E) = \emptyset$.

**Backwards direction.** Suppose (1) $\mathcal{S}(E) \cap Mig_\Gamma(E) = \emptyset$. Let (2) $E' \in Mig_\Gamma(E)$ be given. We have two cases: either $E' = E$ or $E' \neq E$.

Let us consider the case $E' \neq E$. From (1) and (2) we have (3) $E' \notin \mathcal{S}(E)$. From (2) we have (4) $E \sqsubseteq E'$ and (5) $\exists T' : \Gamma \vdash E' : T'$. From Theorem 3.4 we have that (6) $\{\ E''\ \mid\ E \sqsubseteq E'' \sqsubseteq E'\ \}$ is finite. From (6) and Theorem 3.6, we have that there must exist (7) $E'' \in \mathcal{S}(E)$ such that (8) $E \sqsubseteq E'' \sqsubseteq E'$. From (5), (8), and the static gradual guarantee (Theorem 3.3), we have that (9) $\exists T'' : \Gamma \vdash E'' : T''$. From (8) and (9), we have that (10) $E'' \in Mig_\Gamma(E)$. However, together (7) and (10) say that $E'' \in \mathcal{S}(E) \cap Mig_\Gamma(E)$, which contradicts (1). So we conclude that the case $E' \neq E$ is impossible. This leaves only the case $E' = E$, which means that $Mig_\Gamma(E) = \{E\}$, which is a singleton. □

**Putting it all together.** Our singleton checker works as follows:

Algorithm: **Singleton Checker**.

Instance: $E, \Gamma$, where $FV(E) \subseteq Dom(\Gamma)$, where $Mig_\Gamma(E) \neq \emptyset$.

Problem: Is $Mig_\Gamma(E)$ a singleton?

Method:

1. `boolean singleton = true`
2. `for` $(E' \in \mathcal{S}(E))$ {
3. `if` $(\exists T' : \Gamma \vdash E' : T')$ {
4. `singleton = false`
5. }
6. }

7. `return singleton`

Notice that we can verify the assumption $Mig_\Gamma(E) \neq \emptyset$ by checking that $\exists T' : \Gamma \vdash E : T'$.

**Theorem 3.8.** *Algorithm* ***Singleton Checker*** *returns* `true` *iff* $Mig_\Gamma(E)$ *is a singleton.*

*Proof.* We will go through the algorithm step by step.

Step 1: we declare a Boolean variable `singleton` with the initial value `true`. The idea is that unless we find evidence of a second element of $Mig_\Gamma(E)$, aside from $E$ itself, the algorithm will return `true`.

Step 2: we have from Theorem 3.7 that $Mig_\Gamma(E)$ is a singleton iff $\mathcal{S}(E) \cap Mig_\Gamma(E) = \emptyset$. So, we must check that in the body of the `for`-loop the algorithm sets `singleton` to `false` iff at least one $E' \in \mathcal{S}(E)$ has the property that $\exists T' : \Gamma \vdash E' : T'$.

Steps 3–4: if we find $E'$ such that $\exists T' : \Gamma \vdash E' : T'$, then we set `singleton` to `false`. □

**Theorem 3.9.** *We can solve the singleton problem in* $O(n^2)$ *time.*

*Proof.* We have from Theorem 3.8 that Algorithm **Singleton Checker** is correct. We can generate $\mathcal{S}(E)$ in linear time in the size of $E$. The size of $\mathcal{S}(E)$ is linear in the size of $E$. Thus, the algorithm runs a check of $O(n)$ cases that each takes $O(n)$ time, for a total of $O(n^2)$ time. □

## 3.4 The Finiteness Problem

Our algorithm for the finiteness problem uses constraints. Specifically, our algorithm represents the set of possible migrations as the set of solutions to constraints that are generated from the program. Then, our algorithm decides whether the set of solutions is finite.

### 3.4.1 Constraints

We use $v$ to range over a set of type variables *TypeVar*. Define the set *TypeExp* of type expressions $\tau$ as follows.

$$\tau \ ::= \ \texttt{Dyn} \mid \texttt{bool} \mid \texttt{int} \mid \tau \to \tau \mid v$$

We define a class of constraints over type expressions. A constraint is of one of the following four forms:

| | |
|---|---|
| $T \sqsubseteq v$ | Precision constraints |
| $v \rhd v' \to v''$ | Matching constraints |
| $\tau = \tau'$ | Equality constraints |
| $\tau \sim \tau'$ | Consistency constraints |

A *constraint system* is a pair $A = (V, S)$, where $V$ is a finite set of type variables, and $S$ is a set of constraints in which all the type variables are members of $V$. Intuitively, a set of constraints $S$ represents the *conjunction* of the constraints in $S$. Define $vars(A) = V$.

We need notation for talking about different sets of systems of constraints:

| | |
|---|---|
| *PMEC* | if $(V, S) \in PMEC$, then $S$ can contain Precision, Matching, Equality, and Consistency constraints |
| *MEC* | if $(V, S) \in MEC$, then $S$ can contain Matching, Equality, and Consistency constraints |
| *EC* | if $(V, S) \in EC$, then $S$ can contain Equality and Consistency constraints |
| $C$ | if $(V, S) \in C$, then $S$ can contain Consistency constraints |
| $C^-$ | if $(V, S) \in C^-$, then $S$ can contain any Consistency constraint of the form $v \sim \tau$. |

Those five sets of sets of constraints have the following relationships:

$$PMEC \supseteq MEC \supseteq EC \supseteq C \supseteq C^{-}$$

We use $\varphi$ to range over mappings from a finite set of type variables to types. We use $Dom(\varphi)$ to denote the domain of $\varphi$. For a type expression $\tau$, we use $\varphi(\tau)$ to denote $\tau$ in which every variable $v$ has been replaced by $\varphi(v)$. We order mappings as follows:

$$\varphi \leq \varphi' \quad \Longleftrightarrow \quad Dom(\varphi) = Dom(\varphi') \ \wedge \ \forall v \in Dom(\varphi) : \varphi(v) \sqsubseteq \varphi'(v)$$

Notice that $\leq$ is a partial order. If $A = (V, S)$ is a constraint system, then we say that a mapping $\varphi$ from $V$ to types is a *solution* of $A$ if the following conditions are satisfied.

| For each: | we have: |
|---|---|
| $T \sqsubseteq v$ | $T \sqsubseteq \varphi(v)$ |
| $v \rhd v' \to v''$ | $\varphi(v) \rhd \varphi(v') \to \varphi(v'')$ |
| $\tau = \tau'$ | $\varphi(\tau) = \varphi(\tau')$ |
| $\tau \sim \tau'$ | $\varphi(\tau) \sim \varphi(\tau')$ |

Let $Sol(A)$ denote the set of solutions of $A$.

### 3.4.2 Generating Constraints

From $E, \Gamma$, we generate constraints $Gen(E, \Gamma) \in PMEC$ as follows. Assume that $E$ has been $\alpha$-converted so that all bound variables are distinct from each other and distinct from the free variables. Let $X$ be the set of $\lambda$-variables $x$ occurring in $E$, and let $Y$ be a set of variables disjoint from $X$ consisting of a variable $[\![F]\!]$ for every occurrence of the subterm $F$ in $E$. Let

$Z$ be a set of variables disjoint for $X$ and $Y$ consisting of a variable $\langle G \rangle$ for every occurrence of the subterm $(F\ G)$ in $E$. The notations $[\![F]\!]$ and $\langle G \rangle$ are ambiguous because there may be more than one occurrence of some subterm $F$ in $E$ or some subterm $G$ in $E$. However, it will always be clear from context which occurrence is meant. Now we generate the following constraints.

| For every occurrence in $E$ of a subterm of this form: | generate this constraint: |
|---:|---|
| `true` | $[\![\texttt{true}]\!] = \texttt{bool}$ |
| `false` | $[\![\texttt{false}]\!] = \texttt{bool}$ |
| $n$ | $[\![n]\!] = \texttt{int}$ |
| (free variable) $x$ | $[\![x]\!] = \Gamma(x)$ |
| (bound variable) $x$ | $[\![x]\!] = x$ |
| $\lambda x : S.F$ | $[\![\lambda x : S.F]\!] = x \to [\![F]\!] \ \ \wedge \ \ S \sqsubseteq x$ |
| $F\ G$ | $[\![F]\!] \rhd \langle G \rangle \to [\![FG]\!] \ \ \wedge \ \ \langle G \rangle \sim [\![G]\!]$ |

Before we state that the above reduction is correct, we introduce some helper notation. We define let $Dom(\Gamma)$ denote the domain of $\Gamma$: $Dom(\emptyset) = \emptyset$ and $Dom(\Gamma, x : T) = Dom(\Gamma) \cup \{x\}$. We let $FV(E)$ denote the set of free variables of $E$: $FV(n) = \emptyset$ and $FV(True) = \emptyset$ and $FV(False) = \emptyset$ and $FV(x) = \{x\}$ and $FV(\lambda x : T.F) = FV(F) \setminus \{x\}$ and $FV(E_1\ E_2) = FV(E_1) \cup FV(E_2)$.

**Theorem 3.10.** $\forall E, \Gamma :$ *if* $FV(E) \subseteq Dom(\Gamma)$*, then* $(Mig_\Gamma(E), \sqsubseteq)$ *and* $(Sol(Gen(E, \Gamma)), \leq)$ *are order-isomorphic.*

We prove Theorem 3.10 in Appendix A of the supplementary material.

### 3.4.3 Solving Constraints

Our algorithm for solving the finiteness problem uses four transformations that successively transform the constraints to use fewer forms of constraints. Intuitively, the transformations work as follows:

$$PMEC \overset{SimPrec}{\longrightarrow} MEC \overset{SimMatch}{\longrightarrow} EC \overset{SimEq}{\longrightarrow} C \overset{SimCon}{\longrightarrow} C^{-}$$

**Precision constraints.** We define a simplification procedure *SimPrec* that transforms every Precision constraint into zero, one, or more Equality constraints:

$$SimPrec \;:\; PMEC \to MEC$$

We define *SimPrec* to leave the set of type variables unchanged, and to proceed by repeating the following transformation until it no longer has an effect.

| From | To |
|---|---|
| $\texttt{Dyn} \sqsubseteq v$ | (no constraint) |
| $\texttt{bool} \sqsubseteq v$ | $v = \texttt{bool}$ |
| $\texttt{int} \sqsubseteq v$ | $v = \texttt{int}$ |
| $T' \to T'' \sqsubseteq v$ | $v = v' \to v'' \;\wedge\; T' \sqsubseteq v' \;\wedge\; T'' \sqsubseteq v''$ where $v', v''$ are fresh type variables |

**Theorem 3.11.** $\forall A \in PMEC : Sol(A) = Sol(SimPrec(A))$.

*Proof.* Straightforward. □

**Matching constraints.** We define a simplification procedure *SimMatch* that replaces each Matching constraint with one or three Equality constraints. We will use $M$ to refer to the set of Matching constraints. We will use $match(A)$ to refer to the subset of Matching constraints in $A$.

$$SimMatch \quad : \quad MEC \times 2^M \to EC$$

Specifically, for $S \subseteq match(A)$, we define $SimMatch(A, S)$ by

replacing each $(v \rhd v' \to v'') \in A \cap S$ with $(v = v' \to v'')$, and
replacing each $(v \rhd v' \to v'') \in A \setminus S$ with $(v = \texttt{Dyn}) \ \wedge \ (v' = \texttt{Dyn}) \ \wedge \ (v'' = \texttt{Dyn})$.

Intuitively, the role of $S$ is to decide what to do with each matching constraint. Each matching constraint $(v \rhd v' \to v'')$ in $S$ should be turned into an equality constraint $(v = v' \to v'')$. Any other matching constraint $(v \rhd v' \to v'')$ should be turned into the three constraints $(v = \texttt{Dyn}) \ \wedge \ (v' = \texttt{Dyn}) \ \wedge \ (v'' = \texttt{Dyn})$.

Additionally, we define *SimMatch* to leave the set of type variables unchanged.

**Theorem 3.12.** $\forall A \in MEC : Sol(A) = \bigcup_{S \subseteq match(A)} \ Sol(SimMatch(A, S))$.

*Proof.* Straightforward from the definition of $\rhd$. □

**Theorem 3.13.** $\forall A \in MEC : Sol(A)$ *is finite iff* $\forall S \subseteq match(A) : Sol(SimMatch(A, S))$ *is finite.*

*Proof.* Immediate from Theorem 3.12. □

**Equality constraints.** We define the set *Subst* of substitutions that have domain *TypeVar* and range *TypeExp*. For a substitution $\sigma \in Subst$, we define $Dom(\sigma)$ to be the set of type variables $v$ such that $\sigma(v) \neq v$. We use $\sigma \cup \sigma'$ to denote the union of two substitutions $\sigma, \sigma'$ that have disjoint domains.

We define a function *Unify* that solves the Equality constraints.

$$Unify \quad : \quad EC \rightarrow (Subst \cup \{fail\})$$

We define $Unify(A)$ to produce the most general unifier (MGU) of the Equality constraints in $A$, or, if no solution exists, return *fail*.

We define a function *SimEq* that uses a substitution to transform away all Equality constraints.

$$SimEq \quad : \quad (EC \times Subst) \rightarrow C$$

We define $SimEq(A, \sigma)$ as follows. First, the set of type variables is $vars(A) \setminus Dom(\sigma)$. Second, the set of constraints consists of only Consistency constraints: apply the substitution to the Consistency constraints in $A$ and return only those transformed Consistency constraints.

**Theorem 3.14.**

$$\forall A \in EC : Sol(A) = \begin{cases} \{\ (\sigma \circ \sigma') \cup \sigma' \ \mid \ \sigma' \in Sol(SimEq(A, \sigma))\ \} & \text{if } \sigma \neq fail \\ \emptyset & \text{if } \sigma = fail \end{cases}$$

*where* $\sigma = Unify(A)$.

*Proof.* In the case of $\sigma \neq \mathit{fail}$, we have that $\mathit{Dom}(\sigma)$ and $\mathit{Dom}(\sigma') = \mathit{vars}(A) \setminus \mathit{Dom}(\sigma)$ are disjoint so $(\sigma \circ \sigma') \cup \sigma'$ is well defined. Additionally, we have $\sigma = \mathit{Unify}(A)$ so any solution of $A$ when restricted to $\mathit{Dom}(\sigma') = \mathit{vars}(A)$ must equal an element $\sigma'$ of $\mathit{Sol}(\mathit{SimEq}(A, \sigma))$. We can recover any such solution by combining $\sigma'$ with $(\sigma \circ \sigma')$ which replaces any variable $v$ in the codomain of $\sigma$ with $\sigma'(v)$.

In the case of $\sigma = \mathit{fail}$, we have that a subset of $A$ is unsolvable, so $A$ is unsolvable, too, hence $\mathit{Sol}(A) = \emptyset$. □

**Theorem 3.15.** *$\forall A \in EC : \mathit{Sol}(A)$ is finite iff ($\sigma \neq \mathit{fail}$ implies $\mathit{Sol}(\mathit{SimEq}(A, \sigma))$ is finite), where $\sigma = \mathit{Unify}(A)$.*

*Proof.* Immediate from Theorem B.4.3. □

**Consistency constraints.** We define a function *SimCon* that simplifies a set of consistency constraints.

$$\mathit{SimCon} \quad : \quad C \to (C^{-} \cup \{\mathit{fail}\})$$

We define *SimCon* by repeatedly applying the following transformations until no transformation applies. When the To-column lists (*fail*), the entire transformation returns *fail*.

| From | To |
|---|---|
| $\texttt{bool} \sim \texttt{bool}$ | $\emptyset$ |
| $\texttt{int} \sim \texttt{int}$ | $\emptyset$ |
| $\tau \sim \texttt{Dyn}$ | $\emptyset$ |
| $\texttt{Dyn} \sim \tau$ | $\emptyset$ |
| $(\tau_1 \to \tau_2) \sim \texttt{bool}$ | $(\mathit{fail})$ |
| $(\tau_1 \to \tau_2) \sim \texttt{int}$ | $(\mathit{fail})$ |
| $\texttt{bool} \sim (\tau_1 \to \tau_2)$ | $(\mathit{fail})$ |
| $\texttt{int} \sim (\tau_1 \to \tau_2)$ | $(\mathit{fail})$ |
| $\texttt{bool} \sim \texttt{int}$ | $(\mathit{fail})$ |
| $\texttt{int} \sim \texttt{bool}$ | $(\mathit{fail})$ |
| $(\tau_1 \to \tau_2) \sim (\tau_1' \to \tau_2')$ | $\{\ (\tau_1 \sim \tau_1'),\ \ (\tau_2 \sim \tau_2')\ \}$ |
| $\tau \sim v$ | $\{\ v \sim \tau\ \}$ |

**Theorem 3.16.**

$$\forall A \in C : \mathit{Sol}(A) = \begin{cases} \mathit{Sol}(\mathit{SimCon}(A)) & \text{if } \mathit{SimCon}(A) \neq \mathit{fail} \\ \emptyset & \text{otherwise} \end{cases}$$

*Proof.* Straightforward. □

**Boundedness.** We introduce the core concept in our approach to solving the finiteness problem. The idea is to check whether a constraint system is *bounded*, which we will define in three steps.

First, we define a notion of a *path* in a type expression. For a type expression $\tau$, we can create a syntax tree in which every node is labeled by a member of $\{\texttt{Dyn}, \texttt{bool}, \texttt{int}, \to\} \cup$

*TypeVar*. For each node in the syntax tree for $\tau$, we can consider the path $\alpha$ that leads from the root to that node. We use $\tau(\alpha)$ to denote label of the node reached by $\alpha$. We define $\mathit{paths}(\tau)$ to be the set of paths from the root of $\tau$ to all leafs of $\tau$. We have that $\tau$ is finite so also $\mathit{paths}(\tau)$ is finite.

Second, we define a predicate *BoundedVar* on a type variable, a type, and a constraint system. The type is of a special form: each of its leaves is either `bool` or `int`, which means that it is maximal in the precision order. We use *MaximumType* to denote the set of such types.

$$\begin{aligned} \mathit{BoundedVar} &: (\mathit{TypeVar} \times \mathit{MaximumType} \times C^{-}) \to \mathit{Boolean} \\ \mathit{BoundedVar}(v, T, A) &= \forall \alpha \in \mathit{paths}(T) : \exists (v \sim \tau') \in A : \tau'(\alpha) = T(\alpha) \end{aligned}$$

Intuitively, $\mathit{BoundedVar}(v, T, A)$ says that "the variable $v$ is bounded by a $\sqsubseteq$-maximum type $T$ that can be pieced together from constraints in $A$". Specifically, for every leaf in $T$, we require that $A$ contains a constraint $(v \sim \tau')$ such that $\tau'$ has the same type as $T$ at the corresponding leaf. For example, suppose $A = (V, S)$, where

$$\begin{aligned} V &= \{\ v, v_1, v_2\ \} \\ S &= \{\ v \sim (\texttt{bool} \to v_1), \quad v \sim (v_2 \to \texttt{int})\ \} \end{aligned}$$

We have $\mathit{BoundedVar}(v, T, A)$, where $T = \texttt{bool} \to \texttt{int}$. We can see this via a cases analysis, one for each leaf of $T$. For the leaf `bool` of $T$, we have in $A$ the constraint $v \sim (\texttt{bool} \to v_1)$, where $(\texttt{bool} \to v_1)$ has the same type (`bool`) as $T$ at the corresponding leaf. Similarly, for the leaf `int` of $T$. we have in $A$ the constraint $v \sim (v_2 \to \texttt{int})$, where $(v_2 \to \texttt{int})$ has the same type (`int`) as $T$ at the corresponding leaf.

Third, we define a predicate *Bounded* on elements of $C^-$:

$$\begin{aligned} \mathit{Bounded} &: C^- \to \mathit{Boolean} \\ \mathit{Bounded}(A) &= \forall v \in \mathit{vars}(A) : \exists T \in \mathit{MaximumType} : \mathit{BoundedVar}(v, T, A) \end{aligned}$$

The *Bounded* predicate checks whether every solution must assign every variable a type that is bounded by a certain $\sqsubseteq$-maximum type.

**Theorem 3.17.** *For $A \in C^-$: $\mathit{Sol}(A)$ is finite iff $\mathit{Bounded}(A)$.*

*Proof.* In the forwards direction, suppose $\mathit{Sol}(A)$ is finite. Thus, we can pick a maximal $\varphi \in \mathit{Sol}(A)$. Let $v \in \mathit{vars}(A)$. We will define a $T$ such that $\mathit{paths}(T) = \mathit{paths}(\varphi(v))$. Let $\alpha \in \mathit{paths}(\varphi(v))$. Given that $\varphi$ is maximal, the constraints must force $\varphi(v)(\alpha) \in \{\texttt{Dyn}, \texttt{bool}, \texttt{int}\}$. The only way this is possible is that we can find a constraint $(v \sim \tau') \in A$ such that $\tau'(\alpha) \in \{\texttt{bool}, \texttt{int}\}$. So we can define the leaf in $T$ at the end of path $\alpha$ to be $\tau'(\alpha)$. As a result, we have $T \in \mathit{MaximalType}$ and $\mathit{BoundedVar}(v, T, A)$. We conclude $\mathit{Bounded}(A)$.

In the backwards direction, suppose $\mathit{Bounded}(A)$. For each variable, we have a lower bound and upper bound on the types that we can assign that variable. So, from Theorem 3.4 we have that $\mathit{Sol}(A)$ is finite. □

**Theorem 3.18.** *For $A \in C^-$, we can run $\mathit{Bounded}(A)$ in polynomial time.*

*Proof.* We can check $\mathit{Bounded}(A)$ by, for each $v \in \mathit{vars}(A)$, checking whether we can construct $T \in \mathit{MaximumType}$ such that $\mathit{BoundedVar}(v, T, A)$. We do this as follows.

First we collect all constraints in $A$ of the form $(v \sim \tau')$. Let $\bigcup_{\tau'} \mathit{paths}(\tau')$ denote the union of $\mathit{paths}(\tau')$ across all such $\tau'$. Notice that we can construct this union in polynomial time. We see that $\bigcup_{\tau'} \mathit{paths}(\tau')$ defines the largest potential tree shape of $T$. For each $\alpha \in \bigcup_{\tau'} \mathit{paths}(\tau')$

we can determine and record in polynomial time whether for any constraint $(v \sim \tau')$ we have $\tau'(\alpha) \in \{\texttt{bool}, \texttt{int}\}$. Now we can do a tree traversal of the tree shape defined by $\bigcup_{\tau'} \mathit{paths}(\tau')$ and determine whether any subset of $\bigcup_{\tau'} \mathit{paths}(\tau')$ defines a $T \in \mathit{MaximumType}$. This traversal can be done in polynomial time.

In summary, for each of the polynomially many $v \in \mathit{vars}(A)$, we do a polynomial-time check, which gives a total of polynomial time. □

**Putting it all together.** Our finiteness checker works as follows:

Algorithm: **Finiteness Checker**.

Instance: $E, \Gamma$, where $FV(E) \subseteq \mathit{Dom}(\Gamma)$.

Problem: Is $\mathit{Mig}_\Gamma(E)$ finite?

Method:

1. $\mathit{PMEC}\ A_1 = \mathit{Gen}(E, \Gamma)$
2. $\mathit{MEC}\ A_2 = \mathit{SimPrec}(A_1)$
3. `boolean finite` = `true`
4. `for` $(S \subseteq \mathit{match}(A_2))$ {
5. $\quad \mathit{EC}\ A_5 = \mathit{SimMatch}(A_2, S)$
6. $\quad (\mathit{Subst} \cup \{\mathit{fail}\})\ \sigma = \mathit{Unify}(A_5)$
7. $\quad$ `if` $(\sigma \neq \mathit{fail})$ {
8. $\quad\quad C\ A_8 = \mathit{SimEq}(A_5, \sigma)$
9. $\quad\quad (C^- \cup \{\mathit{fail}\})\ A_9 = \mathit{SimCon}(A_8)$
10. $\quad\quad$ `if` $(A_9 \neq \mathit{fail})$ {
11. $\quad\quad\quad$ `if` $\neg \mathit{Bounded}(A_9)$ {
12. $\quad\quad\quad\quad$ `finite` = `false`
13. $\quad\quad\quad$ }

14. }
15. }
16. }
17. `return finite`

Notice that we use as type annotations the names of the five sets of sets of constraints, namely $PMEC$, $MEC$, $EC$, $C$, $C^-$. We also use $(Subst \cup \{fail\})$ and $(C^- \cup \{fail\})$ as type annotations. We can check easily that the algorithm type checks.

**Theorem 3.19.** *Algorithm* ***Finiteness Checker*** *returns* `true` *iff* $Mig_\Gamma(E)$ *is finite.*

*Proof.* We will go through the algorithm step by step.

Step 1: we have from Theorem 3.10 that $(Mig_\Gamma(E), \sqsubseteq)$ and $(Sol(Gen(E,\Gamma)), \leq)$ are order-isomorphic. So, $(Mig_\Gamma(E), \sqsubseteq)$ is finite iff $(Sol(Gen(E,\Gamma)), \leq) = Sol(A_1)$ is finite.

Step 2: we have from Theorem 3.11 that $Sol(A_1) = Sol(SimPrec(A_1)) = Sol(A_2)$.

Step 3: we declare a Boolean variable `finite` with the initial value `true`. The idea is that unless we find evidence of infinitely many solutions, the algorithm will return `true`.

Step 4: we have from Theorem 3.13 $Sol(A_2)$ is finite iff $\forall S \subseteq match(A_2)$ : $Sol(SimMatch(A_2, S))$ is finite. So, we must check that in the body of the `for`-loop the algorithm sets `finite` to `false` iff at least one $S \subseteq match(A_2)$ has the property that $Sol(SimMatch(A_2, S))$ is infinite. We will check this as we go through Steps 5-12.

Step 5: here we consider one of the cases of $S \subseteq match(A_2)$. The goal is to check that the algorithm sets `finite` to `false` iff $Sol(SimMatch(A_2, S)) = Sol(A_5)$ is infinite.

Steps 6–8: we have from Theorem 3.15 that $Sol(A_5)$ is infinite iff $Sol(SimEq(A_5, \sigma)) = Sol(A_8)$ is infinite, where $\sigma = Unify(A_5)$ and $\sigma \neq fail$.

Steps 9–10: we have from Theorem 3.16 that either $SimCon(A_8)$ returns *fail*, in which case we have $Sol(SimCon(A_8)) = \emptyset$, which is finite, and otherwise $Sol(A_8) = Sol(SimCon(A_8)) = Sol(A_9)$.

Steps 11-12: we have from Theorem 3.17 that $Sol(A_9)$ is finite iff $Bounded(A_9)$. Thus, for an execution arrives at Steps 11-12, we have that $Sol(SimMatch(A_2, S)) = Sol(A_5)$, which is finite iff $Sol(A_8) = Sol(A_9)$ is finite. So the algorithm sets `finite` to `false` in exactly the right cases. □

**Theorem 3.20.** *We can solve the finiteness problem in EXPTIME.*

*Proof.* We have from Theorem 3.19 that Algorithm **Finiteness Checker** is correct. Let us analyze the algorithm's time complexity. Let $n$ be the total size of $E$ and $\Gamma$. Step 1 uses polynomial time to construct $A_1 = Gen(E, \Gamma)$, and the size of $A_1$ is $O(n)$. Step 2 uses polynomial time to construct $A_2 = SimPrec(A_1)$, and the size of $A_2$ is $O(n)$. Step 4 is a loop that runs $O(2^n)$ iterations because $match(A_2)$ is of size $O(n)$. Step 5 constructs $A_5$ which is of size $O(n)$. Step 6 constructs $\sigma = Unify(A_5)$ which is of size $O(2^n)$, due to a well-known property of unification. Step 8 constructs $A_8 = SimEq(A_5, \sigma)$ which is of size $O(2^n)$. Step 9 constructs $A_9 = SimCon(A_8)$ which is of size $O(2^n)$. Step 11 runs $Bounded(A_9)$ in time that is polynomial in $O(2^n)$, due to Theorem 3.18. From the rule $(2^a)^b = 2^{ab}$ we have that in total Step 11 runs in time that is $O(2^n)$.

In summary, we have a loop that runs $O(2^n)$ iterations that each takes $O(2^n)$ time. The grand total is $O(2^n) \times O(2^n) = O(2^n)$ which is in EXPTIME. □

### 3.4.4 Example of How Our Finiteness Checker Works: $\lambda x.x(\texttt{succ}(x(\texttt{true})))$

$$
\begin{aligned}
E &= \lambda x.x(\texttt{succ}(x(\texttt{true}))) \\
\Gamma &= [\ \texttt{succ} : \texttt{int} \to \texttt{int}\ ]
\end{aligned}
$$

First we construct $Gen(E, \Gamma)$:

| | |
|---|---|
| $\lambda x.x(\texttt{succ}(x(\texttt{true})))$ | $[\![\lambda x.x(\texttt{succ}(x(\texttt{true})))]\!] = x \to [\![x(\texttt{succ}(x(\texttt{true})))]\!]$ |
| | $\texttt{Dyn} \sqsubseteq x$ |
| $x(\texttt{succ}(x(\texttt{true})))$ | $[\![x]\!] \rhd \langle \texttt{succ}(x(\texttt{true}))\rangle \to [\![x(\texttt{succ}(x(\texttt{true})))]\!]$ |
| | $\langle \texttt{succ}(x(\texttt{true}))\rangle \sim [\![\texttt{succ}(x(\texttt{true}))]\!]$ |
| $x$ | $[\![x]\!] = x$ |
| $\texttt{succ}(x(\texttt{true}))$ | $[\![\texttt{succ}]\!] \rhd \langle x(\texttt{true})\rangle \to [\![\texttt{succ}(x(\texttt{true}))]\!]$ |
| | $\langle x(\texttt{true})\rangle \sim [\![x(\texttt{true})]\!]$ |
| $\texttt{succ}$ | $[\![\texttt{succ}]\!] = \Gamma(\texttt{succ})$ |
| $x(\texttt{true})$ | $[\![x]\!] \rhd \langle \texttt{true}\rangle \to [\![x(\texttt{true})]\!]$ |
| | $\langle \texttt{true}\rangle \sim [\![\texttt{true}]\!]$ |
| $x$ | $[\![x]\!] = x$ |
| $\texttt{true}$ | $[\![\texttt{true}]\!] = \texttt{bool}$ |

Notice that the listing above has two occurrences of $[\![x]\!] = x$. Viewed as a set, $Gen(E, \Gamma)$ consists of 12 constraints. Notice also that in the constraint for $\texttt{succ}$, we can use that $\Gamma(\texttt{succ}) = \texttt{int} \to \texttt{int}$.

Next we apply *SimPrec* to $Gen(E, \Gamma)$. This step removes $\texttt{Dyn} \sqsubseteq x$, which leaves us with the following 10 constraints.

| | |
|---|---|
| $\lambda x.x(\texttt{succ}(x(\texttt{true})))$ | $[\![\lambda x.x(\texttt{succ}(x(\texttt{true})))]\!] = x \to [\![x(\texttt{succ}(x(\texttt{true})))]\!]$ |
| $x(\texttt{succ}(x(\texttt{true})))$ | $[\![x]\!] \rhd \langle \texttt{succ}(x(\texttt{true}))\rangle \to [\![x(\texttt{succ}(x(\texttt{true})))]\!]$ |
| | $\langle \texttt{succ}(x(\texttt{true}))\rangle \sim [\![\texttt{succ}(x(\texttt{true}))]\!]$ |
| $x$ | $[\![x]\!] = x$ |
| $\texttt{succ}(x(\texttt{true}))$ | $[\![\texttt{succ}]\!] \rhd \langle x(\texttt{true})\rangle \to [\![\texttt{succ}(x(\texttt{true}))]\!]$ |
| | $\langle x(\texttt{true})\rangle \sim [\![x(\texttt{true})]\!]$ |
| $\texttt{succ}$ | $[\![\texttt{succ}]\!] = \texttt{int} \to \texttt{int}$ |
| $x(\texttt{true})$ | $[\![x]\!] \rhd \langle \texttt{true}\rangle \to [\![x(\texttt{true})]\!]$ |
| | $\langle \texttt{true}\rangle \sim [\![\texttt{true}]\!]$ |
| $\texttt{true}$ | $[\![\texttt{true}]\!] = \texttt{bool}$ |

Let us use $A_{10}$ to denote the above set of 10 constraints. In the listing of $A_{10}$, we have three Matching constraints, which for brevity of notation, we will number from 1 to 3, as follows:

$$
\begin{aligned}
1 \;&:\; [\![x]\!] \rhd \langle \texttt{succ}(x(\texttt{true}))\rangle \to [\![x(\texttt{succ}(x(\texttt{true})))]\!] \\
2 \;&:\; [\![\texttt{succ}]\!] \rhd \langle x(\texttt{true})\rangle \to [\![\texttt{succ}(x(\texttt{true}))]\!] \\
3 \;&:\; [\![x]\!] \rhd \langle \texttt{true}\rangle \to [\![x(\texttt{true})]\!]
\end{aligned}
$$

Now we must consider all subsets of $\{1, 2, 3\}$. For each $S \subseteq \{1, 2, 3\}$, we must determine whether $SimMatch(A_{10}, S)$ has finitely many solutions.

Let us focus on $S = \{1, 2, 3\}$ and construct $SimMatch(A_{10}, \{1, 2, 3\})$:

| | |
|---|---|
| $\lambda x.x(\texttt{succ}(x(\texttt{true})))$ | $[\![\lambda x.x(\texttt{succ}(x(\texttt{true})))]\!] = x \to [\![x(\texttt{succ}(x(\texttt{true})))]\!]$ |
| $x(\texttt{succ}(x(\texttt{true})))$ | $[\![x]\!] = \langle \texttt{succ}(x(\texttt{true}))\rangle \to [\![x(\texttt{succ}(x(\texttt{true})))]\!]$ |
| | $\langle \texttt{succ}(x(\texttt{true}))\rangle \sim [\![\texttt{succ}(x(\texttt{true}))]\!]$ |
| $x$ | $[\![x]\!] = x$ |
| $\texttt{succ}(x(\texttt{true}))$ | $[\![\texttt{succ}]\!] = \langle x(\texttt{true})\rangle \to [\![\texttt{succ}(x(\texttt{true}))]\!]$ |
| | $\langle x(\texttt{true})\rangle \sim [\![x(\texttt{true})]\!]$ |
| $\texttt{succ}$ | $[\![\texttt{succ}]\!] = \texttt{int} \to \texttt{int}$ |
| $x(\texttt{true})$ | $[\![x]\!] = \langle \texttt{true}\rangle \to [\![x(\texttt{true})]\!]$ |
| | $\langle \texttt{true}\rangle \sim [\![\texttt{true}]\!]$ |
| $\texttt{true}$ | $[\![\texttt{true}]\!] = \texttt{bool}$ |

Notice that the only change from $A_{10}$ to $SimMatch(A_{10}, \{1, 2, 3\})$ is that three occurrences of $\rhd$ turned into $=$.

Next we apply *SimEq* to $SimMatch(A_{10}, \{1,2,3\})$. Notice that $SimMatch(A_{10}, \{1,2,3\})$ has 7 Equality constraints. Those 7 Equality constraints are satisfiable and have the following most general unifier ($\varphi_{123}$), where $p, q$ are type variables.

| $v$ | : | $\varphi_{123}(v)$ |
|---|---|---|
| $[\![\lambda x.x(\texttt{succ}(x(\texttt{true})))]\!]$ | : | $(p \to q) \to q$ |
| $x$ | : | $p \to q$ |
| $[\![x(\texttt{succ}(x(\texttt{true})))]\!]$ | : | $q$ |
| $[\![x]\!]$ | : | $p \to q$ |
| $\langle \texttt{succ}(x(\texttt{true}))\rangle$ | : | $p$ |
| $[\![\texttt{succ}]\!]$ | : | $\texttt{int} \to \texttt{int}$ |
| $\langle x(\texttt{true})\rangle$ | : | $\texttt{int}$ |
| $[\![\texttt{succ}(x(\texttt{true}))]\!]$ | : | $\texttt{int}$ |
| $\langle \texttt{true}\rangle$ | : | $p$ |
| $[\![x(\texttt{true})]\!]$ | : | $q$ |
| $[\![\texttt{true}]\!]$ | : | $\texttt{bool}$ |

Let us use $A'$ to denote the subset of 3 Consistency constraints in $SimMatch(A_{10}, \{1,2,3\})$, which is:

$$\begin{aligned} \langle \texttt{succ}(x(\texttt{true}))\rangle &\sim [\![\texttt{succ}(x(\texttt{true}))]\!] \\ \langle x(\texttt{true})\rangle &\sim [\![x(\texttt{true})]\!] \\ \langle \texttt{true}\rangle &\sim [\![\texttt{true}]\!] \end{aligned}$$

Next we apply $\varphi_{123}$ to $A'$. The result is that $SimEq(SimMatch(A_{10}, \{1,2,3\}), \varphi_{123})$ is:

$$\begin{aligned} p &\sim \texttt{int} \\ \texttt{int} &\sim q \\ p &\sim \texttt{bool} \end{aligned}$$

Let us use $A_{123}$ to denote the above set of 3 Consistency constraints.

Next we apply *SimCon* to $A_{123}$. The effect is to change $\texttt{int} \sim q$ into $q \sim \texttt{int}$:

$$\begin{aligned} p &\sim \texttt{int} \\ q &\sim \texttt{int} \\ p &\sim \texttt{bool} \end{aligned}$$

Let us use $A_{cm}$ to denote the above set of 3 Consistency constraints. We observe that $Bounded(A_{cm})$. Now we use Theorem 3.17 to conclude that $Sol(A_{cm})$ is finite.

Let us return to the step in which we consider different subsets of the Matching constraints. Above we showed that $SimMatch(A_{10}, \{1,2,3\})$ is finite. Now let us focus on $S = \emptyset$ and construct $SimMatch(A_{10}, \emptyset)$:

| | |
|---|---|
| $\lambda x.x(\texttt{succ}(x(\texttt{true})))$ | $[\![\lambda x.x(\texttt{succ}(x(\texttt{true})))]\!] = x \to [\![x(\texttt{succ}(x(\texttt{true})))]\!]$ |
| $x(\texttt{succ}(x(\texttt{true})))$ | $[\![x]\!] = \texttt{Dyn}$ |
| | $\langle \texttt{succ}(x(\texttt{true})) \rangle = \texttt{Dyn}$ |
| | $[\![x(\texttt{succ}(x(\texttt{true})))]\!] = \texttt{Dyn}$ |
| | $\langle \texttt{succ}(x(\texttt{true})) \rangle \sim [\![\texttt{succ}(x(\texttt{true}))]\!]$ |
| $x$ | $[\![x]\!] = x$ |
| $\texttt{succ}(x(\texttt{true}))$ | $[\![\texttt{succ}]\!] = \texttt{Dyn}$ |
| | $\langle x(\texttt{true}) \rangle = \texttt{Dyn}$ |
| | $[\![\texttt{succ}(x(\texttt{true}))]\!] = \texttt{Dyn}$ |
| | $\langle x(\texttt{true}) \rangle \sim [\![x(\texttt{true})]\!]$ |
| $\texttt{succ}$ | $[\![\texttt{succ}]\!] = \texttt{int} \to \texttt{int}$ |
| $x(\texttt{true})$ | $[\![x]\!] = \texttt{Dyn}$ |
| | $\langle \texttt{true} \rangle = \texttt{Dyn}$ |
| | $[\![x(\texttt{true})]\!] = \texttt{Dyn}$ |
| | $\langle \texttt{true} \rangle \sim [\![\texttt{true}]\!]$ |
| $\texttt{true}$ | $[\![\texttt{true}]\!] = \texttt{bool}$ |

Next we apply *SimEq* to $SimMatch(A_{10}, \emptyset)$. Notice that $SimMatch(A_{10}, \emptyset)$ has 13 Equality constraints. Those 13 Equality constraints are unsatisfiable because of two constraints $[\![\texttt{succ}]\!] = \texttt{Dyn}$ and $[\![\texttt{succ}]\!] = \texttt{int} \to \texttt{int}$. So, $Sol(SimMatch(A_{10}, \emptyset)) = \emptyset$.

Let us return to the step in which we consider different subsets of the Matching constraints. From the case of $S = \emptyset$, we see that for sets $S$ where $2 \notin S$, we have that $\mathit{SimMatch}(A_{10}, S)$ contains an unsatisfiable subset of Equality constraints. So, for each of those cases, $\mathit{Sol}(\mathit{SimMatch}(A_{10}, S)) = \emptyset$.

Now let us focus on $S = \{2\}$ and construct $\mathit{SimMatch}(A_{10}, \{2\})$:

| | |
|---|---|
| $\lambda x.x(\texttt{succ}(x(\texttt{true})))$ | $[\![\lambda x.x(\texttt{succ}(x(\texttt{true})))]\!] = x \to [\![x(\texttt{succ}(x(\texttt{true})))]\!]$ |
| $x(\texttt{succ}(x(\texttt{true})))$ | $[\![x]\!] = \texttt{Dyn}$ |
| | $\langle \texttt{succ}(x(\texttt{true})) \rangle = \texttt{Dyn}$ |
| | $[\![x(\texttt{succ}(x(\texttt{true})))]\!] = \texttt{Dyn}$ |
| | $\langle \texttt{succ}(x(\texttt{true})) \rangle \sim [\![\texttt{succ}(x(\texttt{true}))]\!]$ |
| $x$ | $[\![x]\!] = x$ |
| $\texttt{succ}(x(\texttt{true}))$ | $[\![\texttt{succ}]\!] = \langle x(\texttt{true}) \rangle \to [\![\texttt{succ}(x(\texttt{true}))]\!]$ |
| | $\langle x(\texttt{true}) \rangle \sim [\![x(\texttt{true})]\!]$ |
| $\texttt{succ}$ | $[\![\texttt{succ}]\!] = \texttt{int} \to \texttt{int}$ |
| $x(\texttt{true})$ | $[\![x]\!] = \texttt{Dyn}$ |
| | $\langle \texttt{true} \rangle = \texttt{Dyn}$ |
| | $[\![x(\texttt{true})]\!] = \texttt{Dyn}$ |
| | $\langle \texttt{true} \rangle \sim [\![\texttt{true}]\!]$ |
| $\texttt{true}$ | $[\![\texttt{true}]\!] = \texttt{bool}$ |

Next we apply *SimEq* to $\mathit{SimMatch}(A_{10}, \{2\})$. Notice that $\mathit{SimMatch}(A_{10}, \{2\})$ has 11 Equality constraints. Those 11 Equality constraints are satisfiable and have the following most general unifier ($\varphi_2$), where $p, q$ are type variables:

| $v$ | : | $\varphi_2(v)$ |
|---|---|---|
| $[\![\lambda x.x(\texttt{succ}(x(\texttt{true})))]\!]$ | : | $\texttt{Dyn} \to \texttt{Dyn}$ |
| $x$ | : | $\texttt{Dyn}$ |
| $[\![x(\texttt{succ}(x(\texttt{true})))]\!]$ | : | $\texttt{Dyn}$ |
| $[\![x]\!]$ | : | $\texttt{Dyn}$ |
| $\langle \texttt{succ}(x(\texttt{true}))\rangle$ | : | $\texttt{Dyn}$ |
| $[\![\texttt{succ}]\!]$ | : | $\texttt{int} \to \texttt{int}$ |
| $\langle x(\texttt{true})\rangle$ | : | $\texttt{int}$ |
| $[\![\texttt{succ}(x(\texttt{true}))]\!]$ | : | $\texttt{int}$ |
| $\langle \texttt{true}\rangle$ | : | $\texttt{Dyn}$ |
| $[\![x(\texttt{true})]\!]$ | : | $\texttt{Dyn}$ |
| $[\![\texttt{true}]\!]$ | : | $\texttt{bool}$ |

Next we apply $\varphi_2$ to $A'$, The result is that $SimEq(SimMatch(A_{10}, \{2\}), \varphi_2)$ is:

$$\begin{aligned} \texttt{Dyn} &\sim \texttt{int} \\ \texttt{int} &\sim \texttt{Dyn} \\ \texttt{Dyn} &\sim \texttt{bool} \end{aligned}$$

Let us use $A_2$ to denote the above set of 3 Consistency constraints.

Next we apply *SimCon* to $A_2$. The effect is that $SimCon(A_2) = \emptyset$. Finally we observe that $Bounded(\emptyset)$. Now we use Theorem 3.17 to conclude that $Sol(\emptyset)$ is finite.

Let us consider the case of $S = \{1, 2\}$ and construct $SimMatch(A_{10}, \{1, 2\})$:

| | |
|---|---|
| $\lambda x.x(\texttt{succ}(x(\texttt{true})))$ | $[\![\lambda x.x(\texttt{succ}(x(\texttt{true})))]\!] = x \to [\![x(\texttt{succ}(x(\texttt{true})))]\!]$ |
| $x(\texttt{succ}(x(\texttt{true})))$ | $[\![x]\!] = \langle \texttt{succ}(x(\texttt{true}))\rangle \to [\![x(\texttt{succ}(x(\texttt{true})))]\!]$ |
| | $\langle \texttt{succ}(x(\texttt{true}))\rangle \sim [\![\texttt{succ}(x(\texttt{true}))]\!]$ |
| $x$ | $[\![x]\!] = x$ |
| $\texttt{succ}(x(\texttt{true}))$ | $[\![\texttt{succ}]\!] = \langle x(\texttt{true})\rangle \to [\![\texttt{succ}(x(\texttt{true}))]\!]$ |
| | $\langle x(\texttt{true})\rangle \sim [\![x(\texttt{true})]\!]$ |
| $\texttt{succ}$ | $[\![\texttt{succ}]\!] = \texttt{int} \to \texttt{int}$ |
| $x(\texttt{true})$ | $[\![x]\!] = \texttt{Dyn}$ |
| | $\langle \texttt{true}\rangle = \texttt{Dyn}$ |
| | $[\![x(\texttt{true})]\!] = \texttt{Dyn}$ |
| | $\langle \texttt{true}\rangle \sim [\![\texttt{true}]\!]$ |
| $\texttt{true}$ | $[\![\texttt{true}]\!] = \texttt{bool}$ |

Next we apply *SimEq* to $SimMatch(A_{10}, \{1, 2\})$. Notice that $SimMatch(A_{10}, \{1, 2\})$ has 9 Equality constraints. Those 9 Equality constraints are unsatisfiable because of two constraints $[\![x]\!] = \langle \texttt{succ}(x(\texttt{true}))\rangle \to [\![x(\texttt{succ}(x(\texttt{true})))]\!]$ and $[\![x]\!] = \texttt{Dyn}$.
So, $Sol(SimMatch(A_{10}, \{1, 2\})) = \emptyset$.

Let us consider the case of $S = \{2, 3\}$ and construct $SimMatch(A_{10}, \{2, 3\})$:

| | |
|---|---|
| $\lambda x.x(\texttt{succ}(x(\texttt{true})))$ | $[\![\lambda x.x(\texttt{succ}(x(\texttt{true})))]\!] = x \to [\![x(\texttt{succ}(x(\texttt{true})))]\!]$ |
| $x(\texttt{succ}(x(\texttt{true})))$ | $[\![x]\!] = \texttt{Dyn}$ |
| | $\langle \texttt{succ}(x(\texttt{true})) \rangle = \texttt{Dyn}$ |
| | $[\![x(\texttt{succ}(x(\texttt{true})))]\!] = \texttt{Dyn}$ |
| | $\langle \texttt{succ}(x(\texttt{true})) \rangle \sim [\![\texttt{succ}(x(\texttt{true}))]\!]$ |
| $x$ | $[\![x]\!] = x$ |
| $\texttt{succ}(x(\texttt{true}))$ | $[\![\texttt{succ}]\!] = \langle x(\texttt{true}) \rangle \to [\![\texttt{succ}(x(\texttt{true}))]\!]$ |
| | $\langle x(\texttt{true}) \rangle \sim [\![x(\texttt{true})]\!]$ |
| $\texttt{succ}$ | $[\![\texttt{succ}]\!] = \texttt{int} \to \texttt{int}$ |
| $x(\texttt{true})$ | $[\![x]\!] = \langle \texttt{true} \rangle \to [\![x(\texttt{true})]\!]$ |
| | $\langle \texttt{true} \rangle \sim [\![\texttt{true}]\!]$ |
| $\texttt{true}$ | $[\![\texttt{true}]\!] = \texttt{bool}$ |

Next we apply *SimEq* to $SimMatch(A_{10}, \{2, 3\})$. Notice that $SimMatch(A_{10}, \{2, 3\})$ has 9 Equality constraints. Those 9 Equality constraints are unsatisfiable because of two constraints $[\![x]\!] = \texttt{Dyn}$ and $[\![x]\!] = \langle \texttt{true} \rangle \to [\![x(\texttt{true})]\!]$. So, $Sol(SimMatch(A_{10}, \{2, 3\})) = \emptyset$.

In summary, we have shown that in every case of $S$, we find that $SimMatch(A_{10}, S)$ is finite.

We conclude that $Sol(Gen(E, \Gamma))$ is finite, which in turn means that $Mig_{\Gamma}(E)$ is finite.

### 3.4.5 Example of How Our Finiteness Checker Works: $\lambda x.xx$

$$\begin{aligned} E &= \lambda x.xx \\ \Gamma &= [\,] \end{aligned}$$

First we construct $Gen(E, \Gamma)$:

$$\begin{array}{r|l} \lambda x.xx & [\![\lambda x.xx]\!] = x \to [\![xx]\!] \\ & \texttt{Dyn} \sqsubseteq x \\ xx & [\![x]\!] \rhd \langle x \rangle \to [\![xx]\!] \\ & \langle x \rangle \sim [\![x]\!] \\ x & [\![x]\!] = x \\ x & [\![x]\!] = x \end{array}$$

Notice that the listing above has two occurrences of $[\![x]\!] = x$. Viewed as a set, $Gen(E, \Gamma)$ consists of 5 constraints.

Next we apply *SimPrec* to $Gen(E, \Gamma)$. This step removes $\texttt{Dyn} \sqsubseteq x$, which leaves us with the following 4 constraints.

$$\begin{array}{r|l} \lambda x.xx & [\![\lambda x.xx]\!] = x \to [\![xx]\!] \\ xx & [\![x]\!] \rhd \langle x \rangle \to [\![xx]\!] \\ & \langle x \rangle \sim [\![x]\!] \\ x & [\![x]\!] = x \end{array}$$

Let us use $A_4$ to denote the above set of 4 constraints. In the listing of $A_3$, we have a single Matching constraint, which for brevity of notation, we will give number 1:

$$1 \quad : \quad [\![x]\!] \rhd \langle x \rangle \to [\![xx]\!]$$

Now we must consider all subsets of $\{1\}$. For each $S \subseteq \{1\}$, we must determine whether $\mathit{SimMatch}(A_4, S)$ has finitely many solutions.

Let us focus on $S = \{1\}$ and construct $\mathit{SimMatch}(A_4, \{1\})$:

$$\begin{array}{r|l} \lambda x.xx & [\![\lambda x.xx]\!] = x \to [\![xx]\!] \\ xx & [\![x]\!] = \langle x \rangle \to [\![xx]\!] \\ & \langle x \rangle \sim [\![x]\!] \\ x & [\![x]\!] = x \end{array}$$

Next we apply *SimEq* to $\mathit{SimMatch}(A_4, \{1\})$. Notice that $\mathit{SimMatch}(A_{10}, \{1\})$ has 3 Equality constraints. Those 3 Equality constraints are satisfiable and have the following most general unifier ($\varphi_1$), where $p, q$ are type variables:

$$\begin{array}{rcl} v & : & \varphi_1(v) \\ \hline [\![\lambda x.xx]\!] & : & (p \to q) \to q \\ [\![xx]\!] & : & q \\ x & : & p \to q \\ [\![x]\!] & : & p \to q \\ \langle x \rangle & : & p \end{array}$$

Let us use $A'$ to denote the subset of 1 Consistency constraint in $SimMatch(A_4, \{1\})$, which is:

$$\langle x \rangle \quad \sim \quad [\![x]\!]$$

Next we apply $\varphi_4$ to $A'$, The result is that $SimEq(SimMatch(A_{10}, \{1, 2, 3\}), \varphi_1)$ is:

$$p \quad \sim \quad p \to q$$

Let us use $A_1$ to denote the above set of 1 Consistency constraint.

Next we apply *SimCon* to $A_1$. The effect is no change: $SimCon(A_1) = A_1$. We observe that $Bounded(A_1)$ is false. Now we use Theorem 3.17 to conclude that $Sol(A_1)$ is infinite.

In Appendix A of the supplementary material, we show that $p \sim (p \to q)$ has no maximal solution, so $\lambda x.xx$ has no maximal migration.

## 3.5 The Top-Choice Problem

The top-choice problem is: given $E, \Gamma$, does $Mig_\Gamma(E)$ have a greatest element? In other words, is $Mig_\Gamma(E)$ finite and does it have a single maximal migration? We begin with the observation that if $Mig_\Gamma(E)$ has a greatest element, then $Mig_\Gamma(E)$ is finite.

**Theorem 3.21.** *If $Mig_\Gamma(E)$ has a greatest element, then $Mig_\Gamma(E)$ is finite.*

*Proof.* Suppose $Mig_\Gamma(E)$ has a greatest element $E_g$, which means that any migration $E'$ must satisfy $E \sqsubseteq E' \sqsubseteq E_g$. Thus, $Mig_\Gamma(E) \subseteq \{\ E' \mid E \sqsubseteq E' \sqsubseteq E_g\ \}$. We have from Theorem 3.4 that $\{\ E' \mid E \sqsubseteq E' \sqsubseteq E_g\ \}$ is finite so also $Mig_\Gamma(E)$ is finite. □

Given Theorem 3.21, our algorithm for solving the top-choice problem begins with checking that $Mig_\Gamma(E)$ is finite. We do this with the finiteness checker that we presented in Section 3.4. Next we explore $Mig_\Gamma(E)$ and look for elements $E'$ that are maximal elements $Mig_\Gamma(E)$, that is, $Mig_\Gamma(E')$ is a singleton. We do this with the singleton checker that we presented in Section 3.3.

**Putting it all together.** Our top-choice checker works as follows:

Algorithm: **Top-Choice Checker**.

Instance: $E, \Gamma$, where $FV(E) \subseteq Dom(\Gamma)$.

Problem: Does $Mig_\Gamma(E)$ have a greatest element?

Method:

1. `int numMax` $= 0$
2. `if` ($Mig_\Gamma(E)$ is finite) {
3. $\quad 2^{Terms}$ `workset` $= \{E\}$
4. $\quad 2^{Terms}$ `done` $= \emptyset$
5. $\quad$ `while` (`workset` $\neq \emptyset$) {
6. $\quad\quad$ pick $E' \in$ `workset`
7. $\quad\quad$ remove $E'$ from `workset` and add $E'$ to `done`
8. $\quad\quad$ `if` ($\exists T' : \Gamma \vdash E' : T'$) {
9. $\quad\quad\quad$ `if` ($Mig_\Gamma(E')$ is a singleton) {
10. $\quad\quad\quad\quad$ `numMax` $=$ `numMax + 1`
11. $\quad\quad\quad$ } `else` {
12. $\quad\quad\quad\quad$ add ($\mathcal{S}(E') \setminus$ `done`) to `workset`
13. $\quad\quad\quad$ }
14. $\quad\quad$ }
15. $\quad$ }

16. }

17. `return (numMax == 1)`

**Theorem 3.22.** *Algorithm* ***Top-Choice Checker*** *returns* `true` *iff* $Mig_\Gamma(E)$ *has a greatest element.*

*Proof.* We will go through the algorithm step by step.

Step 1: we declare an integer variable `numMax` that holds the number of maximal elements of $Mig_\Gamma(E)$ encountered so far.

Step 2: we check that $Mig_\Gamma(E)$ is finite because otherwise $Mig_\Gamma(E)$ has no greatest element. Additionally, the finiteness check ensures that the search space is finite.

Steps 3–4: we declare two sets of terms, called `workset` and `done`. The idea is classical: `workset` contains terms that we must process, while `done` holds terms that we have already processed.

Step 5: we will iterate until `workset` is done. This is guaranteed to terminate because of the finiteness check in Step 2.

Steps 6–7: we pick a term $E$' to process and update `workset` and `done` accordingly.

Step 8: we check that $E'$ type checks; otherwise $E' \notin Mig_\Gamma(E)$.

Steps 9–12: we check whether $E'$ is a maximal element of $Mig_\Gamma(E)$. If so, then we increase `numMax` by 1, and otherwise use $\mathcal{S}(E')$ to add terms that are one step above $E'$ to `workset`, except for those that we have processed already.

Step 17: if we found a single maximal element, then that element is the greatest element, and we return `true`. Otherwise, we return `false`. □

**Theorem 3.23.** *We can solve the top-choice problem in EXPTIME.*

*Proof.* We have from Theorem 3.22 that Algorithm **Top-Choice Checker** is correct. We have from Section 3.4 that the search space is bounded by a term of a size that is exponential in the size of the input. The total number of terms between $E$ and that bound is exponential in the size of the input. For each term in the search space, we do an amount of work that is polynomial in the size of the term. In summary, the algorithm runs in EXPTIME. □

## 3.6 The Maximality Problem

The question of whether the maximality problem is decidable remains an open problem. In this section we will give a semi-algorithm for the maximality problem and we will show that the problem is NP-hard.

### 3.6.1 A Semi-algorithm for the Maximality Problem

We can adapt the top-choice checker in Section 3.5 to become a semi-algorithm for the maximality problem. We make two modifications, as follows.

First, we skip the check of finiteness. This will ensure that we may find maximal migrations for any input program, but may also make the modified algorithm fail to terminate on some inputs.

Second, we make the algorithm output the maximal migrations, rather than merely counting them.

This semi-algorithm works well for our microbenchmarks: whenever maximal migrations exist, our algorithm finds at least one of them. In practice, we stop the algorithm at a given level of the migration space.

We can adapt the top-choice checker based on programmer needs. For example, we can stop the search based on the desired length of the type annotations, based on the time available, or based on the number of migrations that we want to inspect.

### 3.6.2 The Maximality Problem Is NP-hard

**Theorem 3.24.** *The maximality problem is NP-hard.*

*Proof.* We will do a polynomial-time reduction from 3SAT to the maximality problem. Let

$$F \;=\; \bigwedge_{i=1}^{m} l_{i1} \vee l_{i2} \vee l_{i3}$$

be a formula in which each $l_{ij}$ is either a Boolean variable $x_k$ or its negation $\bar{x}_k$, where $k \in 1..n$. From $F$, we construct the following $\lambda$-term $E_F$ and type environment $\Gamma_F$:

$$\begin{aligned}
E_F \;=\;& \lambda v_1 : (\texttt{Dyn} \to \texttt{int}).\ \ldots\ \lambda v_m : (\texttt{Dyn} \to \texttt{int}). \\
& (v_1 v_1) + \ldots + (v_m v_m) + \\
& ([\lambda \bar{x}_1 : \texttt{Dyn}.(\lambda y_1 : \texttt{int}.\ \bar{x}_1)((v_{g_{11}} \bar{x}_1) + \ldots + (v_{g_{1\overline{m}_1}} \bar{x}_1))] \\
& \qquad ([\lambda x_1 : \texttt{Dyn}.(\lambda z_1 : \texttt{int}.\ x_1)((v_{f_{11}} x_1) + \ldots + (v_{f_{1m_1}} x_1))]\ \texttt{true})) + \ldots + \\
& ([\lambda \bar{x}_n : \texttt{Dyn}.(\lambda y_n : \texttt{int}.\ \bar{x}_n)((v_{g_{n1}} \bar{x}_n) + \ldots + (v_{g_{n\overline{m}_n}} \bar{x}_n))] \\
& \qquad ([\lambda x_n : \texttt{Dyn}.(\lambda z_n : \texttt{int}.\ x_n)((v_{f_{n1}} x_n) + \ldots + (v_{f_{nm_n}} x_n))]\ \texttt{true})) \\
\Gamma_F \;=\;& [\ + \ :\ \texttt{int} \to \texttt{int} \to \texttt{int}\ ]
\end{aligned}$$

The environment $\Gamma_F$ assigns a type to the binary operator $+$; we write uses of $+$ in infix notation. The program $E_F$ has a variable $v_i$ for every clause in $F$, and it has an expression that binds $x_k$ and $\bar{x}_k$ for every variable $x_k$ in $F$.

We use the following notation in the definition of $E_F$. If a variable $x_k$ occurs $m_k$ times in $F$, we use $f_{kp}$ to denote the index of the clause that contains the $p^{th}$ occurrence. Similarly, if a variable $\bar{x}_k$ occurs $\bar{m}_k$ times in $F$, we use $g_{kp}$ to denote the index of the clause that contains the $p^{th}$ occurrence.

The size of $E_F$ is linear in the size of $F$, and we can map $F$ to $E_F$ in polynomial time in a straightforward manner. Additionally, we can check easily that

$$\Gamma_F \vdash E_F : (\texttt{Dyn} \to \texttt{int}) \to \ldots \to (\texttt{Dyn} \to \texttt{int}) \to \texttt{int}$$

The idea of $E_F$ is that the expressions $(v_i\ v_i)$ cause $E_F$ to have no maximal migration, unless the other expressions change that. Specifically, we showed in Section 3.4.5 that $\lambda x.x\ x$ has no maximal migration; here each $(v_i\ v_i)$ plays the role of $(x\ x)$. So, for $E_F$ to have a maximal migration, we need the other expressions to put a bound on every $v_i$. This happens exactly when $F$ is satisfiable.

We will show the following property: $F$ is satisfiable iff $E_F$ has a maximal migration.

Let us consider $Gen(E_F, \Gamma)$ (Section 3.4.2). In essence, $Gen(E_F, \Gamma)$ has three interesting subsets.

First, for each $v_i$, we have in $E_F$ the expression $(v_i v_i)$. This expression ensures that any type of $v_i$, is of the form $r_i \to \texttt{int}$ where $r_i \sim (r_i \to \texttt{int})$ (see Section 3.4.5).

Second, for each literal in the $i$'th clause in $F$, which is $l_{i1} \vee l_{i2} \vee l_{i3}$, we have in $E_F$ the expression $(v_i l_{ij})$. This expression generates the constraint $r_i \sim l_{ij}$ (see the constraint generation rule for application in Section 3.4.2).

Third, for a variable $x_k$ we have in $E_F$ the expression

$$\ldots + ([\lambda \bar{x}_n : \texttt{Dyn}.(\lambda y_n : \texttt{int}.\ \bar{x}_n)(\ldots)]([\lambda x_n : \texttt{Dyn}.(\lambda z_n : \texttt{int}.\ x_n)(\ldots)]\ \texttt{true}))$$

The backbone of this expression is $\ldots + ([\lambda \bar{x}_n : \texttt{Dyn}.\bar{x}_n]([\lambda x_n : \texttt{Dyn}.x_n]\ \texttt{true}))$. This expression generates the constraints $\texttt{bool} \sim x_k \sim \bar{x}_k \sim \texttt{int}$ (see Appendix A of the supplementary material). This is shorthand for the three constraints $(\texttt{bool} \sim x_k)$ and $(x_k \sim \bar{x}_k)$ and $(\bar{x}_k \sim \texttt{int})$. The above expression ensures that we cannot have at the same time that $x_k$ is $\texttt{bool}$ and that $\bar{x}_k$ is $\texttt{int}$.

**Forwards direction.** Suppose $F$ is satisfiable and let $\psi$ be a solution of $F$. Define $\varphi$ as follows:

For each $x_k$ such that $\psi(x_k) = \texttt{true}$, define $\varphi(x_k) = \texttt{bool}$ and $\varphi(\bar{x}_k) = \texttt{Dyn}$.

For each $x_k$ such that $\psi(x_k) = \texttt{false}$, define $\varphi(x_k) = \texttt{Dyn}$ and $\varphi(\bar{x}_k) = \texttt{int}$.

For each $v_i$, define $\varphi(v_i) = \texttt{Dyn} \to \texttt{int}$.

We will show that $\varphi$ is a maximal solution of $Gen(E_F, \Gamma)$. First we show that $\varphi$ is a solution. We will consider, in turn, each of the three interesting subsets of $Gen(E_F, \Gamma)$. (1) Given that $\varphi(v_i) = \texttt{Dyn} \to \texttt{int}$, we have $\varphi(r_i) = \texttt{Dyn}$. So, we have that $\varphi \models r_i \sim (r_i \to \texttt{int})$. (2) For a constraint $r_i \sim l_{ij}$, we have $\varphi(r_i) = \texttt{Dyn}$, so $\varphi \models r_i \sim l_{ij}$. (3) For a variable $x_k$, we can check easily that $\varphi \models \texttt{bool} \sim x_k \sim \bar{x}_k \sim \texttt{int}$.

Second we will show that $\varphi$ is maximal.

Consider $x_k$. Notice that while one of $\varphi(x_k)$ and $\varphi(\bar{x}_k)$ is $\texttt{Dyn}$, we cannot replace that $\texttt{Dyn}$ with anything larger because of the constraint $\texttt{bool} \sim x_k \sim \bar{x}_k \sim \texttt{int}$.

Consider $v_i$. From that $F$ is satisfiable, we have that we can find $j \in \{1,2,3\}$ such that $\varphi(l_{ij}) \in \{\texttt{bool}, \texttt{int}\}$. From the constraint $r_i \sim l_{ij}$, we have that $\varphi(r_i) \in \{\texttt{Dyn}, \texttt{bool}, \texttt{int}\}$. We also have the constraint $r_i \sim (r_i \to \texttt{int})$, so we see that we must have $\varphi(r_i) = \texttt{Dyn}$. We conclude that we cannot replace $\varphi(v_i)$ with anything larger.

**Backwards direction.** Suppose $Gen(E_F, \Gamma)$ has a maximal solution $\varphi$. Define a mapping $\psi$ as follows.

$$\psi(x) \;=\; \begin{cases} \texttt{true} & \text{if } \varphi(x) = \texttt{bool} \\ \texttt{false} & \text{if } \varphi(x) = \texttt{Dyn} \end{cases}$$

We will show that $\psi$ satisfies $F$. Consider the $i$'th clause of $F$. Given that $\varphi$ is a maximal solution of $Gen(E_F, \Gamma)$, we have that $\varphi(v_i)$ is constrained by something, which must happen in a constraint of the form $r_i \sim l_{ij}$. Given that $\varphi$ is a maximal solution of $Gen(E_F, \Gamma)$, we know that, for each $x$, the mapping $\varphi$ assigns either $[\varphi(x) = \texttt{bool}$ and $\varphi(\bar{x}) = \texttt{Dyn}]$, or $[\varphi(x) = \texttt{Dyn}$ and $\varphi(\bar{x}) = \texttt{int}]$. So, we can find $j \in \{1,2,3\}$ such that $\varphi(l_{ij}) \in \{\texttt{bool}, \texttt{int}\}$. We have two cases.

First, suppose $l_{ij}$ is $x$. From $\varphi(l_{ij}) = \varphi(x) \in \{\texttt{bool}, \texttt{int}\}$, we get $\varphi(x) = \texttt{bool}$, hence $\psi(x) = \texttt{true}$, which means that $\psi$ satisfies the $i$'th clause.

Second, suppose $l_{ij}$ is $\bar{x}$. From $\varphi(l_{ij}) = \varphi(\bar{x}) \in \{\texttt{bool}, \texttt{int}\}$, we get $\varphi(\bar{x}) = \texttt{int}$, hence $\varphi(x) = \texttt{Dyn}$, hence $\psi(x) = \texttt{false}$, which means that $\psi$ satisfies the $i$'th clause. □

### 3.6.3 Example of How the NP-hardness Proof Works

$$F_2 \;=\; (x_1 \vee \bar{x}_2 \vee x_3) \;\wedge\; (\bar{x}_1 \vee x_2 \vee x_3)$$

From $F_2$, we construct the following $\lambda$-term $E_{F_2}$ and type environment $\Gamma$:

$$
\begin{array}{rcl}
E_{F_2} & = & \lambda v_1 : (\mathtt{Dyn} \to \mathtt{int}).\ \lambda v_2 : (\mathtt{Dyn} \to \mathtt{int}). \\
& & (v_1 v_1) + (v_2 v_2) + \\
& & ([\lambda \bar{x}_1 : \mathtt{Dyn}.(\lambda y_1 : \mathtt{int}.\ \bar{x}_1)(v_2 \bar{x}_1)] \\
& & \qquad ([\lambda x_1 : \mathtt{Dyn}.(\lambda z_1 : \mathtt{int}.\ x_1)(v_1 x_1)]\ \mathtt{true})) + \\
& & ([\lambda \bar{x}_2 : \mathtt{Dyn}.(\lambda y_2 : \mathtt{int}.\ \bar{x}_2)(v_1 \bar{x}_2)] \\
& & \qquad ([\lambda x_2 : \mathtt{Dyn}.(\lambda z_2 : \mathtt{int}.\ x_2)(v_2 x_2)]\ \mathtt{true})) + \\
& & ([\lambda \bar{x}_3 : \mathtt{Dyn}.(\lambda y_n : \mathtt{int}.\ \bar{x}_3)\ 0] \\
& & \qquad ([\lambda x_3 : \mathtt{Dyn}.(\lambda z_3 : \mathtt{int}.\ x_3)((v_1 x_3) + (v_2 x_3))]\ \mathtt{true})) \\
\Gamma & = & [\ + \ :\ \mathtt{int} \to \mathtt{int} \to \mathtt{int}\ ]
\end{array}
$$

Notice the use of 0 in $E_{F_2}$; it signals an empty sum that stems from that $\bar{x}_3$ does not occur in $F_2$.

We have that $F_2$ is satisfiable and we will show that $Gen(E_{F_2}, \Gamma)$ has a maximal solution. Here are the three interesting subsets of $Gen(E_{F_2}, \Gamma)$:

$$
\begin{array}{llll}
r_1 \sim (r_1 \to \mathtt{int}) & r_1 \sim x_1 & r_2 \sim \bar{x}_1 & \mathtt{bool} \sim x_1 \sim \bar{x}_1 \sim \mathtt{int} \\
r_2 \sim (r_2 \to \mathtt{int}) & r_1 \sim \bar{x}_2 & r_2 \sim x_2 & \mathtt{bool} \sim x_2 \sim \bar{x}_2 \sim \mathtt{int} \\
 & r_1 \sim x_3 & r_2 \sim x_3 & \mathtt{bool} \sim x_3 \sim \bar{x}_3 \sim \mathtt{int}
\end{array}
$$

We see that all of $x_1, \bar{x}_1, x_2, \bar{x}_2, x_3, \bar{x}_3$ are bounded. Now we turn to $r_1$ and $r_2$. Let focus on a particular satisfying assignment for $F_2$, namely $\psi$ defined as follows: $\psi(x_1) = \texttt{true}$ and $\psi(x_2) = \texttt{true}$ and $\psi(x_3) = \texttt{false}$. From $\psi$ we get $\varphi$:

$$\begin{array}{llll} \varphi(x_1) = \texttt{bool} & \varphi(x_2) = \texttt{bool} & \varphi(x_3) = \texttt{Dyn} & \varphi(v_1) = \texttt{Dyn} \to \texttt{int} = r_1 \to \texttt{int} \\ \varphi(\bar{x}_1) = \texttt{Dyn} & \varphi(\bar{x}_2) = \texttt{Dyn} & \varphi(\bar{x}_3) = \texttt{int} & \varphi(v_2) = \texttt{Dyn} \to \texttt{int} = r_2 \to \texttt{int} \end{array}$$

We can check easily that $\varphi \models \mathit{Gen}(E_{F_2}, \Gamma)$. Additionally, we can check that $\varphi$ is a maximal solution. Let us check every use of $\texttt{Dyn}$. First consider $\varphi(\bar{x}_1) = \texttt{Dyn}$. We see that the constraints $\texttt{bool} \sim x_1 \sim \bar{x}_1 \sim \texttt{int}$ and $\varphi(x_1) = \texttt{bool}$ imply that we must have

$$\texttt{bool} = \varphi(x_1) \sim \bar{x}_1 \sim \texttt{int}$$

Thus, we cannot improve $\varphi(\bar{x}_1) = \texttt{Dyn}$. Similar reasoning applies to the cases of $\varphi(\bar{x}_2) = \texttt{Dyn}$ and $\varphi(x_3) = \texttt{Dyn}$. Next consider $\varphi(v_1) = \texttt{Dyn} \to \texttt{int}$. We see that the constraints $r_1 \sim (r_1 \to \texttt{int})$ and $r_1 \sim x_1$ and that $\varphi(x_1) = \texttt{bool}$ imply that we must have

$$(r_1 \to \texttt{int}) \sim r_1 \sim \varphi(x_1) = \texttt{bool}$$

Thus, we cannot improve $\varphi(v_1) = \texttt{Dyn} \to \texttt{int}$. Similar reasoning applies to the case of $\varphi(v_2) = \texttt{Dyn} \to \texttt{int}$.

We can do a similar analysis of other satisfying assignments for $F_2$ and in each case we will find that $\mathit{Gen}(E_F, \Gamma)$ has a maximal solution.

### 3.6.4 Can the NP-hardness Proof Be Adapted to Other Problems?

For each of the top-choice problem and the finiteness problem, we have an exponential-time upper bound on the time complexity but no lower bound. Let us consider whether the NP-hardness proof for the maximality problem can be adapted to the top-choice problem or the finiteness problem. We make two observations based on the example in Section 3.6.3.

First, if we try other satisfying assignments of $F$ than the $\psi$ that we used in the example, we get other maximal solutions of $\mathit{Gen}(E, \Gamma)$ that are different from $\psi$. So, $\mathit{Mig}_\Gamma(E_F)$ does not have a greatest element, hence the proof is of no help with proving a lower bound for top-choice problem.

Second, consider an assignment $\varphi$ that assigns $\varphi(x_1) = \varphi(\bar{x}_1) = \varphi(x_2) = \varphi(\bar{x}_2) = \varphi(x_3) = \varphi(\bar{x}_3) = \texttt{Dyn}$. This part of the definition of $\varphi$ satisfies most of the constraints in $\mathit{Gen}(E, \Gamma)$ and leaves only $r_1 \sim (r_1 \to \texttt{int})$ and $r_2 \sim (r_2 \to \texttt{int})$. However, those constraints have infinitely many solutions. So, $\mathit{Mig}_\Gamma(E_F)$ has infinitely many solutions, hence the proof is of no help with proving a lower bound for finiteness problem.

## 3.7 Implementation and Experimental Results

**Implementation.** We have implemented our algorithms in Haskell, for a total of 1,159 lines of code. This includes a type checker, a singleton checker, a top-choice checker, a finiteness checker, and a search for a maximal migration. In addition to answers to the singleton, top-choice, and finiteness questions, our tool outputs a maximal migration, if one exists.

| Benchmark | Singleton? | Top Choice? | Finite? | Has Max? |
|---|---|---|---|---|
| $\lambda x.x(\texttt{succ}(x))$ | ✓ | ✓ | ✓ | ✓ |
| $\lambda x.x(\texttt{succ}(x(\texttt{true})))$ | × | ✓ | ✓ | ✓ |
| $\lambda x. + (x\ 4)(x\ \texttt{true})$ | × | ✓ | ✓ | ✓ |
| $(\lambda x.x)4$ | × | ✓ | ✓ | ✓ |
| $\texttt{succ}((\lambda y.y)((\lambda x.x)\texttt{true}))$ | × | × | ✓ | ✓ |
| $\lambda x.x$ | × | × | × | ✓ |
| $\lambda x.\lambda y.yxx$ | × | × | × | ✓ |
| $\lambda x.(\lambda y.x)xx$ | × | × | × | ✓ |
| $\lambda x.(\lambda f.(\lambda x.\lambda y.x)f(fx))(\lambda z.1)$ | × | × | × | ✓ |
| $\lambda x.xx$ | × | × | × | × |
| $(\lambda x.\lambda y.y(xI)(xK))\Delta$ | × | × | × | ? |
| *selfInterpreter* | × | × | × | ? |

**Figure 3.3:** Our benchmarks. Legend: ✓ means *yes*, and × means *no*, and ? means *unknown*.

| Benchmark | Singleton? | Top Choice? | Finite? | Has Max? |
|---|---|---|---|---|
| $\lambda x.x(\texttt{succ}(x))$ | 326 ± 18 ns | 131 ± 2 µs | 126 ± 5 µs | 545 ± 12 ns |
| $\lambda x.x(\texttt{succ}(x(\texttt{true})))$ | 147 ± 5 ns | 313 ± 6 µs | 296 ± 4 µs | 3 ± 1 µs |
| $\lambda x. + (x\ 4)(x\ \texttt{true})$ | 168 ± 3 ns | 533 ± 14 µs | 517 ± 12 µs | 3 ± 1 µs |
| $(\lambda x.x)4$ | 132 ± 6 ns | 35 ± 1 µs | 33 ± 1 µs | 531 ± 32 ns |
| $\texttt{succ}((\lambda y.y)((\lambda x.x)\texttt{true}))$ | 335 ± 3 ns | 209 ± 4 µs | 196 ± 4 µs | 2 µs |
| $\lambda x.x$ | 46 ± 3 ns | 6 ± 1 µs | 6 µs | 371 ± 20 ns |
| $\lambda x.\lambda y.yxx$ | 132 ± 2 ns | 19 ± 1 µs | 19 µs | 2 ms |
| $\lambda x.(\lambda y.x)xx$ | 142 ± 3 ns | 25 µs | 25 ± 1 µs | 2 ± 1 µs |
| $\lambda x.(\lambda f.(\lambda x.\lambda y.x)f(fx))(\lambda z.1)$ | 213 ± 3 ns | 77 ± 2 µs | 77 ± 2 µs | 4 ± 1 ms |
| $\lambda x.xx$ | 88 ns | 8 µs | 8 µs | 2 ms |
| $(\lambda x.\lambda y.y(xI)(xK))\Delta$ | 310 ± 4 ns | 131 ± 3 µs | 131 ± 3 µs | 367 ± 7 ms |
| *selfInterpreter* | 672 ± 15 ns | 587 ± 14 µs | 586 ± 17 µs | 5 s |

**Figure 3.4:** Execution times.

**Benchmarks.** Figure 3.3 shows our benchmarks (column 1) and their key features (columns 2–5): is the set of migrations a singleton, does it have a greatest element, is it finite, and does it have a maximal element?

Notice that the benchmarks include the programs in Figure 2.1. Additionally, the benchmark $(\lambda x.\lambda y.y(xI)(xK))\Delta$ has the curious property that it is strongly normalizing but untypable in System F [GR88, Section 4]. It uses the abbreviations $I = \lambda a.a$, $K = \lambda b.\lambda c.b$, and $\Delta = \lambda d.dd$. Finally, the benchmark *selfInterpreter* is the lambda-term

$$Y[\lambda e.\lambda m.m(\lambda x.x)(\lambda mn.(em)(en))(\lambda m.\lambda v.e(mv))]$$

which is a self-interpreter for pure lambda-calculus [Mog92, Section 3]. It uses the abbreviation $Y = \lambda h.(\lambda x.h(xx))(\lambda x.h(xx))$.

For all benchmarks, we use $\Gamma = [\texttt{succ} : \texttt{int} \to \texttt{int}\ ,\ + : \texttt{int} \to \texttt{int} \to \texttt{int}]$.

**Execution.** We ran each of our tools 100 times or more on each benchmark. Figure 3.4 shows the mean and standard deviation of the timing in each case. We left out the standard deviation in cases where it rounded off to zero. We managed the process with the help of Criterion, a benchmarking tool for Haskell, `http://hackage.haskell.org/package/criterion`.

**Our results.** Our tool answers all the questions in Figure 2.1 correctly, with the footnote that for $\lambda x.xx$, we stopped the exploration at level 5, and for the last two benchmarks, we stopped the exploration at level 4. The early termination is due to that our maximality checker is a semi-algorithm rather than a decision procedure, and for those three programs, no maximal solution exists. Figure 3.5 shows maximal migrations that our tool has given as output. We have used our tool to check that each of those maximal migrations type checks

| Benchmark | Maximal migration |
|---|---|
| $\lambda x.x(\texttt{succ}(x))$ | $\lambda x : \texttt{Dyn}.x(\texttt{succ}(x))$ |
| $\lambda x.x(\texttt{succ}(x(\texttt{true})))$ | $\lambda x : (\texttt{Dyn} \to \texttt{int}).x(\texttt{succ}(x(\texttt{true})))$ |
| $\lambda x. + (x\ 4)(x\ \texttt{true})$ | $\lambda x : (\texttt{Dyn} \to \texttt{int}). + (x\ 4)(x\ \texttt{true})$ |
| $(\lambda x.x)4$ | $(\lambda x : \texttt{int}.x)4$ |
| $\texttt{succ}((\lambda y.y)((\lambda x.x)\texttt{true}))$ | $\texttt{succ}((\lambda y : \texttt{int}.y)((\lambda x : \texttt{Dyn}.x)\texttt{true}))$ |
| $\lambda x.x$ | $\lambda x : \texttt{int}.x$ |
| $\lambda x.\lambda y.yxx$ | $\lambda x : \texttt{int}.\lambda y : (\texttt{int} \to \texttt{int} \to \texttt{int}).yxx$ |
| $\lambda x.(\lambda y.x)xx$ | $\lambda x : \texttt{Dyn}.(\lambda y : \texttt{int}.x)xx$ |
| $\lambda x.(\lambda f.(\lambda x.\lambda y.x)f(fx))(\lambda z.1)$ | $\lambda x : \texttt{int}.(\lambda f : \texttt{Dyn}.(\lambda x : \texttt{int}.\lambda y : \texttt{int}.x)f(fx))$ $(\lambda z : \texttt{int}.1)$ |
| $\lambda x.xx$ | *no maximal migration* |
| $(\lambda x.\lambda y.y(xI)(xK))\Delta$ | *unknown* |
| *selfInterpreter* | *unknown* |

**Figure 3.5:** Our tool's output of maximal migrations.

and is indeed maximal. We do the maximality check by using our singleton checker to check that its set of migrations is a singleton.

**Comparison.** Our tool uses the same input format as the tool that accompanies the chapter by [CCE18]. This enables a head-to-head comparison of our tool and their tool, for our benchmarks. Their tools supports multiple lambda constructors; we used the one called `CDLam`, which provides the most flexibility for migration. We ran their function called `measureMG`, which produces a type for the entire program but outputs no migration. So, we compare the types generated by the two tools, see Figure 3.6. Notice that for every benchmark, the type produced by their tool is $\sqsubseteq$-related to the type produced by our tool. For six benchmarks, the types are different, for three benchmarks the types are the same, and for one benchmark, no maximal migration exists but the tool from [CCE18] produces a type

| Benchmark | POPL 2018 tool | Our tool |
|---|---|---|
| $\lambda x.x(\texttt{succ}(x))$ | $\texttt{Dyn} \to \texttt{Dyn}$ | $\texttt{Dyn} \to \texttt{Dyn}$ |
| $\lambda x.x(\texttt{succ}(x(\texttt{true})))$ | $\texttt{Dyn} \to \texttt{Dyn}$ | $(\texttt{Dyn} \to \texttt{int}) \to \texttt{int}$ |
| $\lambda x. + (x\ 4)(x\ \texttt{true})$ | $\texttt{Dyn} \to \texttt{int}$ | $(\texttt{Dyn} \to \texttt{int}) \to \texttt{int}$ |
| $(\lambda x.x)4$ | $\texttt{Dyn}$ | $\texttt{int}$ |
| $\texttt{succ}((\lambda y.y)((\lambda x.x)\texttt{true}))$ | $\texttt{int}$ | $\texttt{int}$ |
| $\lambda x.x$ | $\texttt{Dyn} \to \texttt{Dyn}$ | $\texttt{int} \to \texttt{int}$ |
| $\lambda x.\lambda y.yxx$ | $\texttt{Dyn} \to \texttt{Dyn} \to \texttt{Dyn}$ | $\texttt{int} \to (\texttt{int} \to \texttt{int} \to \texttt{int}) \to \texttt{int}$ |
| $\lambda x.(\lambda y.x)xx$ | $\texttt{Dyn} \to \texttt{Dyn}$ | $\texttt{Dyn} \to \texttt{Dyn}$ |
| $\lambda x.(\lambda f.(\lambda x.\lambda y.x)f(fx))(\lambda z.1)$ | $\texttt{Dyn} \to \texttt{Dyn}$ | $\texttt{int} \to \texttt{int}$ |
| $\lambda x.xx$ | $\texttt{Dyn} \to \texttt{Dyn}$ | *no maximal migration* |
| $(\lambda x.\lambda y.y(xI)(xK))\Delta$ | $\texttt{Dyn} \to \texttt{Dyn}$ | *unknown* |
| *selfInterpreter* | $\texttt{Dyn}$ | *unknown* |

**Figure 3.6:** The types for the entire program produced by the tool from [CCE18] and by our tool.

anyway. The differences highlight that the tool from [CCE18] may produce non-maximal migrations.

**The reduction.** We have implemented the reduction in Section 3.6.2 from 3SAT to the maximality problem. Each use of the reduction maps a Boolean formula to a lambda-term. We have experimented with mapping several Boolean formulas to lambda-terms and found that in each case, our maximality checker gave the expected result. In particular, the maximality checker found correctly (in 1.11 ms) that $E_{F_2}$ from Section 3.6.3 has a maximal migration. As another example, we tried the following unsatisfiable Boolean formula $F_3$.

$$\begin{aligned} F_3 \quad = \quad & (x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee \bar{x}_3) \wedge (x_1 \vee \bar{x}_2 \vee x_3) \wedge (x_1 \vee \bar{x}_2 \vee \bar{x}_3) \wedge \\ & (\bar{x}_1 \vee x_2 \vee x_3) \wedge (\bar{x}_1 \vee x_2 \vee \bar{x}_3) \wedge (\bar{x}_1 \vee \bar{x}_2 \vee x_3) \wedge (\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3) \end{aligned}$$

We stopped exploration at level 3 (after 478 ms), reflecting that $E_{F_3}$ has no maximal migration.

## 3.8 Related Work

We will discuss related work on type migration in more detail. The most closely related work is the POPL 2018 paper by [CCE18] which we have discussed in Chapter 2. Campora et al. presented an efficient approach to migrating a program, but did not address our four decision problems. We saw in Section 3.7 that the approach in [CCE18] may produce non-maximal migrations; we will give an example of this below. The approach of [CCE18] integrates gradual types and variational types. Specifically, for each $\lambda$-bound variable, the approach uses constraints and unification to produce a static variational type that potentially can replace `Dyn`. If the program has $n$ such variables, those choices between `Dyn` and a static type create a finite migration space of size $2^n$, which the approach searches efficiently.

For example, consider the program $(\lambda x : \texttt{Dyn}.xx)$, which has no maximal migration. The constraint generation procedure generates a constraint of the form $\alpha \approx_\top d\langle \texttt{Dyn}, \alpha\rangle \to \beta$. Here, $\alpha, \beta$ are type variables, $d\langle \texttt{Dyn}, \alpha\rangle$ is a type that signals a choice between `Dyn` and $\alpha$, and $\approx_\top$ is a relationship that must be established via unification. The unification procedure finds that if we pick $\alpha$, then $\alpha = \alpha \to \beta$ has no solution, so the approach picks `Dyn`. As a final step, the approach converts $\beta$ to `Dyn`, and outputs the type $(\texttt{Dyn} \to \texttt{Dyn})$ for $\alpha$, which happens to be the type of the entire term (as shown in Figure 3.6). The example shows that cases where unification fails tend to push the results towards types with more uses of `Dyn`. Notice also that the approach's finite migration space ensures that it always find a migration, even for $(\lambda x : \texttt{Dyn}.xx)$ which has no maximal migration. We found that the approach is of little help with deciding questions such as the maximality problem and the finiteness problem.

[SV08] presented the first algorithm for type migration with gradual types. Their starting point was the type system by [ST06], for which they did type migration with a unification-based algorithm. Later, [GC15] presented a different unification-based algorithm for a similar type system. Both [SV08] and [GC15] proved correctness, while neither had a report on experiments. Those two papers and use similar forms of consistency constraints to ours, but they differ in what questions they answer about such constraints. Specifically, [SV08] and [GC15] focus on finding a single solution, while this chapter studies properties of the set of solutions.

[RCH12] presented a migration algorithm for an object-oriented language with subtyping. They proved that the added types cannot cause new run-time failures.

## 3.9 Summary

We have presented algorithms and a hardness result for deciding key properties of programs in the gradually typed lambda-calculus. Several problems remain open, including whether the maximality problem is decidable, whether the finiteness problem is NP-hard, and whether the top-choice problem can be approached more efficiently than using the finiteness checker as a subroutine.

# CHAPTER 4

# Designing and Migrating Rank-2 Intersection Types

In this chapter, we present a gradual intersection type system that satisfies algorithmic migration criteria of Chapter 2 and some of Siek et al's gradual typing criteria. Our type system is the first gradual intersection type system that has algorithmic support for type migration. We achieve this by using Rank-2 intersection types and by showing that three migration problems, the Singleton, Top Checker and Finiteness problems, have the same time complexities as for the GTLC, which we presented in the proceeding chapter. The Maximality problem remains open.

## 4.1 Introduction

### 4.1.1 Gradual intersection types.

Originally, researchers designed intersection types to characterize normalization properties of lambda-terms [BCD83]. Later, Reynolds introduced intersection types into programming languages in Forsythe [Rey88, Rey96], and they have since been used with a refinement restriction [FP91, DP00, DP03], and broadly building on Reynolds' work, without a refinement restriction [Dun14, OSA16].

Mainstream languages with support for intersection types include Java, TypeScript, and Scala. In more detail, Java has supported intersection types since Java 5 in 2004 [Zar19, Jav22], while TypeScript has supported intersection types since TypeScript 1.6 in 2015 [Sha22, Typ22], and Scala has supported intersection types since Scala 3 in 2021 [Wam20, Sca22]. In Java, a typical use of intersection types is for extending two types without naming a new type. In TypeScript and Scala, a typical use of intersection types is for expressing ad hoc polymorphism.

Typed Racket has supported intersection types since 2015 [KKT15]. Castagna and Lanvin introduced a theoretical model of gradual intersection types [CL17]. Both type systems are highly expressive but neither conforms all of Siek et al's criteria [SVC15a].

Similarly, Castagna et al's gradual intersection types [CLP19] satisfy some of Siek et al's criteria but does not explore the dynamic gradual guarantee.

None of the works have explored algorithmic support for type migration.

### 4.1.2 The Rest of the Chapter.

In Section 4.2 we discuss potential pitfalls and we justify our design. In Sections 4.3–4.4 we present our Rank-2 intersection type system and show that it satisfies some of the criteria from Siek et al. [SVC15a]. In Section 4.5 we show that our type system satisfies our migration criterion, in Section 4.6 we present experimental results, in Section 4.7, we show how to extend our gradual language with further flexibility by adding commutativity and associativity and in Section 4.8 we discuss related work.

## 4.2 Design Motivation

In this section we introduce the running example of the chapter and use it to motivate our design choices. Our goal is to design a language that supports automatic type migration. We obtain such a language by restricting gradual intersection types in a way that facilitates tool support, while simultaneously satisfying some of Siek et al's criteria.

### 4.2.1 Example

Our running example is the program

$$(\lambda x : \texttt{Dyn}.(x\ 4) + (x\ True))(\lambda y : \texttt{Dyn}.5) \tag{4.1}$$

The program executes in a few steps and gives the result 10.

Let us consider type migration for the example. In the GTLC, we can give $x$ a more precise type know that the argument is a function to `Int`:

$$(\lambda x : \texttt{Dyn} \to \texttt{Int}.(x\ 4) + (x\ True))(\lambda y : \texttt{Dyn}.5) \tag{4.2}$$

The program applies $x$ to both an integer and a Boolean. Hence, in a type language with intersections, we can give it the following type:

$$(\lambda x : (\texttt{Int} \to \texttt{Int}) \wedge (\texttt{Bool} \to \texttt{Int}).(x\ 4) + (x\ True))(\lambda y : \texttt{Dyn}.5) \tag{4.3}$$

Here the intersection, expressed by $\wedge$, means that $x$ has both the type $(\texttt{Int} \to \texttt{Int})$ and the type $(\texttt{Bool} \to \texttt{Int})$. Notice that the type of $x$ is fully static and is useful because $x$ is used

twice. Indeed, the type of $x$ says that we use $x$ both as a function of type ($\texttt{Int} \to \texttt{Int}$) and as a function of type ($\texttt{Bool} \to \texttt{Int}$).

In a GTLC extended with our rank-2 intersection types, we cannot make the type of $y$ any more static. The reason is that the argument ($\lambda y : \texttt{Dyn}.5$) has type ($\texttt{Dyn} \to \texttt{Int}$) and no type that is more precise is also *consistent* with both ($\texttt{Int} \to \texttt{Int}$) and ($\texttt{Bool} \to \texttt{Int}$). The notion of consistency is central in gradual typing; As we shall see in the next section, consistency only needs to be defined for simple types. Thus, the definition of consistency is the same as that in Chapter 3.

### 4.2.2 Rank-2 intersection types avoid undecidability.

For full-fledged intersection types, type inference is undecidable [KW04], which seems like a bad omen for type migration with gradual intersection types. We avoid undecidability by using a restricted form of types, namely Rank-2 intersection types [Lei83, Jim95, KW04] for which type inference is EXPTIME-complete [Jim95].

we consider the type migration problem as at least as hard as the type inference problem. Type inference assigns a fully static type to every expression in the program. Type migration in gradually typed systems does not assume that every variable can be assigned a static type. If the underlying type system cannot express the type of a variable, then we must assign it a dynamic type. The migration space is often infinite for a given program, and so unless we would like to assign a `Dyn` type to every variable in a program, migration is a non-trivial task.

### 4.2.3 One type per variable occurrence avoids bloated migration spaces.

Rank-2 intersection types have the potential to bloat the migration space. For example, consider the program $\lambda x : \texttt{Dyn}.xx$, which is untypable in the STLC and has no maximal

migration in the GTLC [MP19]. This program can be migrated in many ways using Rank-2 intersection types, including

$$\lambda x : (\texttt{Dyn} \wedge (\texttt{Dyn} \to \texttt{Dyn})).xx$$

$$\lambda x : (\texttt{Dyn} \wedge (\texttt{Dyn} \to \texttt{Dyn}) \wedge ((\texttt{Dyn} \to \texttt{Dyn}) \to \texttt{Dyn})).xx$$

$$\ldots$$

What we see here is an infinite migration path that bloats the migration space with migrations of little interest.

More generally, consider a program with a type annotation $T$. Non-trivial intersections can be all types that are less precise than $T$, but they are all expected to be valid types, intuitively, because we are merely removing type information from $T$. These conjuncts are not useful however, because they are not carrying new type information. For example, suppose that $T = \texttt{Int} \to \texttt{Int}$. A program with such type annotation should also typecheck with the types: $\texttt{Int} \to \texttt{Dyn}$, $\texttt{Dyn} \to \texttt{Int}$, $\texttt{Dyn} \to \texttt{Dyn}$ and $\texttt{Dyn}$. All of these types could be elements of the conjunction and thus subject to migration themselves. This could result in up to an exponential overhead in the migration space.

We avoid bloat by restricting, for $\lambda x : \sigma.e$, the number of conjuncts in $\sigma$ to be the number of free occurrences of $x$ in $e$, or a single conjunct that would be the type of every free occurrence of $x$ in $e$. If $\sigma$ has a wrong number of conjuncts, we consider $\lambda x : \sigma.e$ to be ill-typed. For example, for the identity function $\lambda x : \sigma.x$, where $x$ occurs a single time, the type $\sigma$ must have a single conjunct. Another example is the program $\lambda x : \sigma.xx$, for which $\sigma$ must have two conjuncts, that is, one for each occurrence of the variable $x$.

The unique-type property of this system pushes us towards having a a simple type rule for a function application $(E_1\ E_2)$: if $E_1$ has type $(\sigma_1 \wedge \sigma_2) \to \sigma'$, and $E_2$ has type $\sigma$, then we will check both that $\sigma_1$ is consistent with $\sigma$ and that $\sigma_2$ is consistent with $\sigma$.

As a technical device, we will label each variable occurrence to help tie it to "its" conjunct in an intersection type. For example, we can label as follows: $\lambda x : (\texttt{Int} \to \texttt{Int}) \wedge \texttt{Int}.x_0\ x_1$. Here, $x_0$ has type $(\texttt{Int} \to \texttt{Int})$, while $x_1$ has type $\texttt{Int}$.

Notice that this restriction results in the loss of commutativity and associativity, but we later extend our system with a limited form of commutativity and associativity. We discuss this in Section 4.7.

### 4.2.4 Our language must satisfy the Static Gradual Guarantee

Our design decision to allow a single conjunct to be the type of every free occurrence of a variable has a side effect: we must disallow idempotence. We previously established that multiple conjuncts in an abstraction annotation correspond to multiple variable occurrences in the abstraction body. So, consider the ill-typed expression $(\lambda x : \texttt{Dyn} \wedge \texttt{Int}.x)$. which has a single occurrence of $x$ but two conjuncts in the type of $x$. If we make the type of $x$ more precise, we can get the expression $(\lambda x : \texttt{Int} \wedge \texttt{Int}.x)$, which is also ill-typed, for the same reason as before. However, if type intersection is idempotent, we would have identities like $\texttt{Int} \wedge \texttt{Int} = \texttt{Int}$, which would mean that $(\lambda x : \texttt{Int} \wedge \texttt{Int}.x)$ can be rewritten as $(\lambda x : \texttt{Int}.x)$, which type checks. Here, idempotence helped get us from an ill-typed expression to a more precise well-typed expression, which violates the static gradual guarantee. The static gradual guarantee ensures that well-typed programs in the original static language remain well-typed in the gradual language, which is an important requirement for gradual languages. Thus, we must disallow idempotence.

While Neergaard and Mairson [NM04] showed that the lack of idempotence can cause type inference to be the same as normalization in intersection types, we do not face this issue in our type-system. Intuitively, the reason this is not an issue is that we consider a syntactic interpretation of gradual types and restrict the types in such a way that they can be captured by our constraints.

### 4.2.5 Soundness and Migratory typing

We discuss the soundness theorem we achieved in this system. Our soundness theorem is weaker than the one for, say, a model of Typed Racket. Specifically, the underlying static type system is essentially the STLC plus some syntactic sugar. The translation from a Rank-2 gradual program into this underlying system must erase gradual types up to a simple type. This approach directly implements the design philosophy outlined of Cimini and Siek [CS16] and others and fails to achieve a traditional soundnesss theorem for this particular extension. Thus, the rank-1 type from our running example would be translated to the rank-0 type $\texttt{Dyn} \to \texttt{Int}$.

Our characterization rests on the results of Greenman and Felleisen [GF18] who establish that, in the context of gradual typing, different type-enforcement systems yield different type-soundness guarantees. Technically speaking, Greenman et al.'s work differentiates three variants of type soundness, in decreasing order of strength:

1. one based on a higher-order enforcement of types, a direct model of Tobin-Hochstadt and Felleisen [TF08] work on Typed Racket;

2. one inspired by transient checking, an idea due to Vitousek et al. [VSS17] and their work on Reticulated Python; and

3. the non-enforcement of types, as popularized by TypeScript [BAT14].

Lazarek et al. [LGF21] also provide empirical evidence that higher-order semantics are not always beneficial in practice.

In addition to weakening the soundness guarantees of our system, our translation also inhibits proper blame tracking in some cases. Consider the program:

$$(\lambda x : \texttt{Int} \to \texttt{Int} \wedge \texttt{Int}.xx)((\lambda y : \texttt{Dyn}.y)\texttt{True}).$$

Traditionally, we should expect that the type-enforcing casts blame the term

$$((\lambda y : \texttt{Dyn}.y)\texttt{True})$$

, because the expression does not yield a value of type int. Since our translation to GTLC yields:

$$(\lambda x : \texttt{Dyn}.xx)((\lambda y : \texttt{Dyn}.y)\texttt{True})$$

However, the blame is instead assigned to the first occurrence of x in the first function, because it is not a function. That is, rather than identifying the outer boundary, the error is caught deep in the resulting target syntax tree.

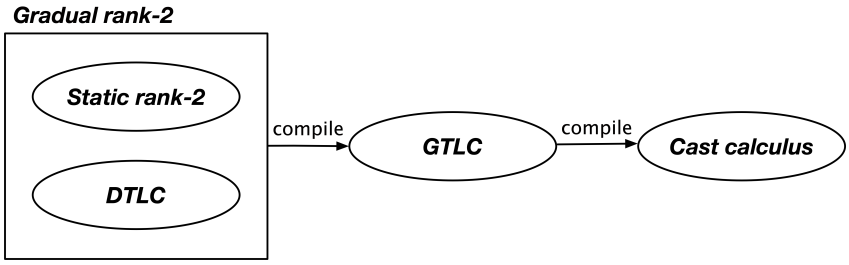


**Figure 4.1:** Our calculus and its relation to four other languages.

## 4.3 Our Type System

We present a gradual Rank-2 intersection type discipline for a $\lambda$-calculus. This design came about via a stepping stone. First we design the static Rank-2 type system and then we *gradualize* it using the methodology of Garcia et al. [GCT16]. The system extends STLC syntactically which helps us with the gradualization procedure, but does not provide practical usability over the STLC so we omit it from the chapter. Appendix B.1 contains the static Rank-2 type system and the gradualization procedure. In outline, this involves defining a concretization function and using it to define type precision and consistency.

We prove in Section 4.4 that our language satisfies some of the criteria from Siek et al. [SVC15a] including a weak type soundness theorem a la TypeScript. Those proofs involve a total of five languages, as illustrated in Figure 4.1. Briefly, our gradual Rank-2 calculus is a conservative extension of both a static Rank-2 calculus and the dynamically typed lambda-calculus (DTLC). We define the semantics of our language by compiling it to the GTLC. The GTLC itself has a semantics that is defined via compilation to a cast calculus.

### 4.3.1 Types and terms.

Figure 4.2 presents our types and terms. Types have three different ranks, as in any Rank-2 type system. Rank-0 types are the types of the GTLC. Any gradual language must contain the `Dyn` type, and here `Dyn` ends up as a Rank-0 type. Rank-1 types are conjunctions of Rank-0 types. Type environments map variables to Rank-1 types, which means that also type annotations are Rank-1 types. Finally, we have Rank-2 types, which we can assign to expressions. A Rank-2 type is either a Rank-0 type or a type of the form $T^1 \to T^2$. We use a notion of rank from Jim [Jim95], and we note that some authors use different definitions of rank, including [Lei83].

Our language is a standard lambda-calculus with booleans and integers. The terms are the same as those of the GTLC [ST06, SV08, SVC15b], in the convenient reformulation by Cimini and Siek [CS16], except for two differences. First, type annotations have Rank-1, as mentioned before. Second, each variable has a label that indicates its order of occurrence in an expression. Those labels enable us to easily implement a design decision from Section 4.2, which is to assign a single type to each variable occurrence. Furthermore, types are labeled with 0, 1 or 2 to indicate their rank. Throughout the chapter, we may skip writing the superscripts when they can be inferred easily from the context.

Figure 4.2 also defines type precision and term precision, which enable us to state and prove the static gradual guarantee.

Type precision is defined by rules that are much like those of the GTLC. The main difference lies in the new rule ($T^1 and$), which says that we can make an intersection type more precise by making each of the conjuncts more precise. This rule is unsurprising given that we have a similar rule ($T^1 T^2 arr$) for function types.

Term precision is defined by rules that are like those of the GTLC.

Syntax:

$$
\begin{array}{rrcl}
\text{(Types)} & \tau^0, \sigma^0, T^0 & ::= & \texttt{Dyn} \mid \texttt{Bool} \mid \texttt{Int} \mid T^0 \to T^0 \\
& \tau^1, \sigma^1, T^1 & ::= & T^0 \mid T^1 \wedge T^1 \\
& \tau^2, \sigma^2, T^2 & ::= & T^0 \mid T^1 \to T^2 \\
& \tau, \sigma, T & ::= & \tau^0 \mid \tau^1 \mid \tau^2 \\
\text{(Terms)} & E & ::= & \texttt{true} \mid \texttt{false} \mid n \mid x_l \mid \lambda x : \sigma^1 . E \mid E\ E \\
\text{(Environments)} & \Gamma & ::= & \emptyset \mid \Gamma, x : \sigma^1 \\
\text{(Labels)} & & ::= & l \in \mathbb{N}
\end{array}
$$

Type precision:

$$\texttt{Dyn} \sqsubseteq T^2 \quad Dyn \sqsubseteq T^1 \quad T^2 \sqsubseteq T^2$$

$$\frac{T_1^1 \sqsubseteq T_1^{1'} \quad T_2^1 \sqsubseteq T_2^{1'}}{T_1^1 \wedge T_2^1 \sqsubseteq T_1^{1'} \wedge T_2^{1'}}\ (T^1 and)$$

$$\frac{T^1 \sqsubseteq T^{1'} \quad T^2 \sqsubseteq T^{2'}}{T^1 \to T^2 \sqsubseteq T^{1'} \to T^{2'}}\ (T^1 T^2 arr)$$

Term precision:

$$E \sqsubseteq E \textit{(P-Refl-E)} \qquad \frac{T_1^1 \sqsubseteq T_2^1 \quad E_1 \sqsubseteq E_2}{\lambda x : T_1^1 . E_1 \sqsubseteq \lambda x : T_2^1 . E_2}\ \textit{(P-Abs)} \qquad \frac{E_1 \sqsubseteq E_3 \quad E_2 \sqsubseteq E_4}{(E_1\ E_2) \sqsubseteq (E_3\ E_4)}\ \textit{(P-App)}$$

**Figure 4.2:** The gradual Rank-2 intersection-typed $\lambda$-calculus: types and terms.

### 4.3.2 Typing Rules.

Figure 4.3 presents our typing rules, along with the helper relations, matching and consistency, which are standard in gradual type systems. Indeed, our definitions of matching and consistency are the same as those in the GTLC [CS16].

We can think of matching as a check to ensure we can represent a type as an arrow type. Clearly this is true for an arrow type, while `Dyn`, after a suitable runtime check, can be represented as $\texttt{Dyn} \to \texttt{Dyn}$. Otherwise, the matching relation is undefined.

Consistency is defined only for Rank-0 types, which is because our type rule for function application assigns the argument a Rank-0 type, as we will explain in more detail below.

Our typing rules use judgments of the form $\Gamma \vdash E : T^2$, where the type environment $\Gamma$ is a function that maps variables to $T^1$ types.

The rules (*T-True*), (*T-False*) and (*T-Num*) are straightforward and standard.

The rule (*T-Var*) assigns a variable occurrence the "right" type from the type environment. The label of each occurrence of $x$ is an index between 1 and $n$, so if $\Gamma$ maps a variable to $\sigma_0^0 \wedge \ldots \wedge \sigma_n^0$, then a variable $x$ with label $i$ gets the type $\sigma_i^0$. We can compare rule (*T-Var*) to the rule from the static Rank-2 intersection type system in [Jim95]:

$$\frac{x : (\bigwedge_{i \in I} \tau_i) \in \Gamma \quad \text{where } i \in I}{\Gamma \vdash x : \tau_i}$$

This rule allows us to assign the variable $x$ any type from the intersection, but it does not give us a unique type for each expression. So we modify the rule to express the idea that we want to assign a type from the intersection to the "right" variable occurrence. Given a variable $x_l$, rule (*T-Var*) grabs the type at position $l$ in the Rank-1 type given to $x$ by $\Gamma$. For

Typing Rules

$\Gamma \vdash E : T^2$.

$\Gamma \vdash \texttt{true} : \texttt{Bool}$ *(T-True)* $\quad$ $\Gamma \vdash \texttt{false} : \texttt{Bool}$ *(T-False)* $\quad$ $\Gamma \vdash n : \texttt{Int}$ *(T-Num)*

$$\frac{x : \bigwedge_{i \in I} \tau_i^0 \in \Gamma \quad l \in I}{\Gamma \vdash x_l : \tau_l^0} \ (T\text{-}Var)$$

$$\frac{\begin{array}{c}\Gamma, (x : \sigma_0^0 \wedge \ldots \wedge \sigma_n^0) \vdash E : \tau \\ where\ \forall i \in 0..n, x_i\ occurs\ in\ E, \\ and\ \sigma_0^0 = \ldots = \sigma_n^0 = \sigma^0\end{array}}{\Gamma \vdash (\lambda x : \sigma^0.E) : \sigma^0 \to \tau} \ (T\text{-}Abs\text{-}0)$$

$$\frac{\Gamma, (x : \sigma_0^0 \wedge ... \wedge \sigma_n^0) \vdash E : \tau \quad where\ \forall i \in 0..n,\ x_i\ occurs\ in\ E}{\Gamma \vdash (\lambda x : \sigma_0^0 \wedge ... \wedge \sigma_n^0.E) : \sigma_0^0 \wedge ... \wedge \sigma_n^0 \to \tau} \ (T\text{-}Abs\text{-}1)$$

$$\frac{\Gamma \vdash E : \tau \quad where\ x_0\ does\ not\ occur\ in\ E}{\Gamma \vdash (\lambda x : \tau^0.E) : \tau^0 \to \tau} \ (T\text{-}Abs\text{-}2)$$

$$\frac{\begin{array}{cc}\Gamma \vdash E_1 : \sigma_1 & \Gamma \vdash E_2 : \sigma^0 \\ \sigma_1 \rhd ((\bigwedge_{i \in I} \tau_i^0) \to \sigma) & \forall i \leq I : \sigma^0 \sim \tau_i^0\end{array}}{\Gamma \vdash E_1\ E_2 : \sigma} \ (T\text{-}App)$$

Consistency:

$\sigma^0 \sim \texttt{Dyn}$ *(C-Dyn1)* $\quad$ $\texttt{Dyn} \sim \sigma^0$ *(C-Dyn2)* $\quad$ $\texttt{Bool} \sim \texttt{Bool}$ *(C-Bool)*

$$\texttt{Int} \sim \texttt{Int} \ (C\text{-}Int)$$

$$\frac{\sigma^0 \sim \sigma'^0 \quad \tau^0 \sim \tau'^0}{(\sigma^0 \to \tau^0) \sim (\sigma'^0 \to \tau'^0)} \ (C\text{-}Arrow)$$

Matching:

$(\sigma^1 \to \tau^2) \rhd (\sigma^1 \to \tau^2)$ *(M-Arrow)* $\quad$ $\texttt{Dyn} \rhd (\texttt{Dyn} \to \texttt{Dyn})$ *(M-Dyn)*

**Figure 4.3:** The gradual Rank-2 intersection-typed $\lambda$-calculus: Typing Rules

example, consider $\Gamma = [x : (\texttt{Int} \wedge \texttt{Bool})]$. Here, the occurrence $x_1$ gets the type `Int` and the occurrence $x_2$ gets the type `Bool`. Thus, our version of the rule supports unique types.

Our type system has three rules for $\lambda$-abstraction: (*T-Abs-0*), (*T-Abs-1*), and (*T-Abs-2*). Our motivation for those rules begins with the single rule for $\lambda$-abstraction from the static Rank-2 intersection type system in [Jim95]:

$$\frac{\Gamma, x : \sigma \vdash E : \tau}{\Gamma \vdash (\lambda x : \sigma . E) : \sigma \to \tau}$$

The above rule covers three distinct possibilities in a uniform manner: $\sigma$ is a $T^0$ type, $\sigma$ is an intersection type, and $x$ does not occur free in $E$. However, in our type system, we want to give a single type for each variable occurrence, which is difficult to achieve with a single rule. In particular, we want to design our type system such that it is an extension of the GTLC. For example, the expression $\lambda x : \texttt{Int} . x + x$ should be well-typed but also the expression $\lambda x : (\texttt{Int} \to \texttt{Int}) \wedge \texttt{Int} . xx$ should be well-typed. So, instead of a single rule, (*T-Abs-0*) covers the case where $\sigma$ is a $T^0$ type, (*T-Abs-1*) covers the case where $\sigma$ is an intersection type, and (*T-Abs-2*) covers the case where $x$ does not occur free in $E$.

The rule (*T-Abs-0*) allows us to type check expressions in the same way as the GTLC. For an expression of the form $\lambda x : \sigma^0 . E$ where x occurs in $E$ a total of $n$ times, when type checking $E$, we append $n$ occurrences of $\sigma^0$ to the type environment as follows $x : (\sigma^0 \wedge ... \wedge \sigma^0)$. By the time we get to type checking a particular occurrence, we can grab the type based on the index. So to type check $\lambda x : \texttt{Int} . x + x$, we type check $x + x$ in the context of $\Gamma = [x : (\texttt{Int} \wedge \texttt{Int})]$.

The rule (*T-Abs-1*) adds a binding of the type annotation to the environment. For example, we can handle $\lambda x : (\texttt{Int} \to \texttt{Int}) \wedge \texttt{Int} . xx$ by appending $x : (\texttt{Int} \to \texttt{Int}) \wedge \texttt{Int}$ to the type environment. The first occurrence of $x$ gets the type $\texttt{Int} \to \texttt{Int}$, and the second occurrence gets the type $\texttt{Int}$.

The rule (*T-Abs-2*) handles the case where a bound variable does not occur in the body of the abstraction at all. In this case, we leave the type environment unchanged when we type check the body of the $\lambda$-abstraction.

Our type system has a single rule (*T-App*) for function application that is quite different from the corresponding rule from the static Rank-2 intersection type system in [Jim95]:

$$\frac{\Gamma \vdash E_1 : ((\bigwedge_{i \in I} \tau_i) \to \sigma) \qquad (\forall i \in I)\ \Gamma \vdash E_2 : \tau_i}{\Gamma \vdash E_1\ E_2 : \sigma}$$

In the above rule, the argument types are Rank-0 types and the rule does not yield unique types. In contrast, our rule implements a design decision from Section 4.2, which is to type check the argument once and thereby get unique types. So, our rule (*T-App*) assigns a single $\sigma^0$ to the argument. If in $(E_1\ E_2)$ we have that the type of $E_1$, after matching, is of the form $T^1 \to \sigma$, we require that every type in the conjunction $T^1$ must be consistent with the argument type. This means that we do not admit the program $(\lambda x : (\texttt{Int} \to \texttt{Bool}) \wedge (\texttt{Bool} \to \texttt{Bool}).x)(\lambda y : \texttt{Int}.y)$. We will however annotate the program as follows: $(\lambda x : (\texttt{Int} \to \texttt{Bool}) \wedge (\texttt{Bool} \to \texttt{Bool}).x)(\lambda y : \texttt{Dyn}.y)$.

### 4.3.3 Example.

Let us demonstrate how the typing rules work by going over the derivation tree for one of our examples, namely $(\lambda x : \tau.(x\ 4) + (x\ True))(\lambda y : \texttt{Dyn}.5)$.

Let $\tau = (\texttt{Int} \to \texttt{Int}) \wedge (\texttt{Bool} \to \texttt{Int})$, let $\tau_1 = \texttt{Int} \to \texttt{Int}$, let $\tau_2 = \texttt{Bool} \to \texttt{Int}$, and let $\Gamma = [+ : \texttt{Int} \to \texttt{Int} \to \texttt{Int},\ \ x : \tau]$.

Let us begin with the type derivation for the subexpression $\lambda x : \tau.(x\ 4) + (x\ True)$, omitting a lookup in $\Gamma$ to get the type of $+$:

$$\dfrac{\dfrac{\dfrac{\Gamma \vdash x_0 : \tau_1 \quad \Gamma \vdash 4 : \texttt{Int}}{\Gamma \vdash x_0\ 4 : \texttt{Int}}\ \textit{T-App} \quad \dfrac{\Gamma \vdash x_1 : \tau_2 \quad \Gamma \vdash True : \texttt{Bool}}{\Gamma \vdash x_1\ True : \texttt{Int}}\ \textit{T-App}}{\Gamma \vdash (x_0\ 4) + (x_1\ True) : \texttt{Int}}\ \textit{T-App}}{\emptyset \vdash \lambda x : \tau.(x_0\ 4) + (x_1\ True)) : \tau \to \texttt{Int}}\ \textit{T-Abs-1}$$

Our full derivation tree is then as follows:

$$\dfrac{\dfrac{\dots}{\emptyset \vdash \lambda x : \tau.(x_0\ 4) + (x_1\ True)) : \tau \to \texttt{Int}} \quad \dfrac{\emptyset \vdash 5 : \texttt{Int}}{\emptyset \vdash \lambda y : \texttt{Dyn}.5 : \texttt{Dyn} \to \texttt{Int}}\ \textit{T-Abs-2}}{\emptyset \vdash (\lambda x : \tau.(x_0\ 4) + (x_1\ True))(\lambda y : \texttt{Dyn}.5) : \texttt{Int}}\ \textit{T-App}$$

Let us discuss a few highlights from this example. First, each variable is labeled based on its occurrence in the expression. When a variable is type checked, we check the label. In the derivation, we have $\Gamma \vdash x_0 : \tau_1$ since $\tau = (\texttt{Int} \to \texttt{Int}) \wedge (\texttt{Bool} \to \texttt{Int})$ and we are interested in the $0^{th}$ element, which is $\tau_1 = \texttt{Int} \to \texttt{Int}$. The second point to note is about abstractions. When we type check the expression $\lambda x : \tau.(x\ 4) + (x\ True)$, we must check that the number of conjuncts in $\tau$ matches the number of variable occurrences. We have two conjuncts in $\tau$ and two variable occurrences. This is checked by *T-Abs-1*. On the other hand, when type-checking the expression $\lambda y : \texttt{Dyn}.5$, we can see that $y$ does not occur in the body, which is why we have a Rank-0 type annotation. This is handled by *T-Abs-2*. Finally, when type checking the top level application, we can see that the function type is $\tau \to \texttt{Int} = ((\texttt{Int} \to \texttt{Int}) \wedge (\texttt{Bool} \to \texttt{Int})) \to \texttt{Int}$ and the argument type is $\texttt{Dyn} \to \texttt{Int}$. Clearly, $(\texttt{Int} \to \texttt{Int}) \sim (\texttt{Dyn} \to \texttt{Int})$ and $(\texttt{Bool} \to \texttt{Int}) \sim (\texttt{Dyn} \to \texttt{Int})$. This is handled by *T-App*.

As shown, the typing rules ensure that the number of conjuncts in a type annotation matches the number of variable occurrences, as we can see from the uses of *T-Var*.

### 4.3.4 Semantics.

Like for GTLC, we define the dynamic semantics of our calculus by a translation to a cast calculus [SVC15a]. Indeed, we use the same cast calculus that researchers have used as the target for translation of GTLC.

Our translation from Rank-2 to the cast calculus proceeds in two steps: first we translate Rank-2 to GTLC, and then we translate GTLC to the cast calculus. The first step is new, while the second step is standard. This level of modularity enables us to rely on known properties of the second step when we prove properties of our semantics. Indeed, for our purposes the key notation from the GTLC and the cast calculus that we need are as follows, all taken from Siek et al. [SVC15a].

We begin with some GTLC notation. For a GTLC type environment $\Gamma$, a GTLC term $E$, and a GTLC type $T$, a GTLC type judgment is of the form $\Gamma \vdash_G E : T$. We let $\sqsubseteq_G$ denote the precision relation in GTLC. For the full definition of the GTLC, see Appendix B.5.

Next we introduce some cast-calculus notation. The calculus extends the STLC with the type `Dyn` and a cast expression $E : (S \Rightarrow^{\mathcal{L}} T)$ that has the following type rule:

$$\frac{\Gamma \vdash_{CC} S}{\Gamma \vdash_{CC} (E : (S \Rightarrow^{\mathcal{L}} T)) : T}$$

The cast expression casts the type from $S$ to $T$, and $\mathcal{L}$ is referred to as the blame label. We use $f \to r$ to denote a step of execution of a cast-calculus term $f$ to $r$. The full definitions can be found in appendix B.5. As usual, $f \to^* r$ denotes zero, one, or more steps of execution.

Now we move on to notation about the semantics of GTLC. The translation of the GTLC to the cast calculus is written $\Gamma \vdash E \rightsquigarrow f : T$, where $\Gamma$ is a GTLC type environment, $E$ is a GTLC term, $f$ is a cast-calculus term, and $T$ is the type of $f$. The semantics of GTLC is given

$$
\begin{array}{rcl@{\qquad}rcl}
\mathcal{T}(\emptyset) & = & \emptyset & \mathcal{T}(n) & = & n \\
\mathcal{T}(\Gamma, x:\sigma) & = & \mathcal{T}(\Gamma), x:\mathcal{T}(\sigma) & \mathcal{T}(True) & = & True \\
 & & & \mathcal{T}(False) & = & False \\
\mathcal{T}(\tau^0) & = & \tau^0 & \mathcal{T}(x_l) & = & x \\
\mathcal{T}(\tau_1 \wedge ... \wedge \tau_n) & = & \tau_1 \sqcap ... \sqcap \tau_n & \mathcal{T}(\lambda x:\tau^1.E) & = & \lambda x:\mathcal{T}(\tau^1).\mathcal{T}(E) \\
\mathcal{T}(\tau_1 \to \tau_2) & = & \mathcal{T}(\tau_1) \to \mathcal{T}(\tau_2) & \mathcal{T}(E_1\ E_2) & = & \mathcal{T}(E_1)\ \mathcal{T}(E_2)
\end{array}
$$

**Figure 4.4:** Compilation from Rank-2 to GTLC.

by first translating to a term in the cast calculus and then executing the cast-calculus term. Specifically, the notation $E \Downarrow_G v$ means that we execute a GTLC term $E$ to a cast-calculus value $v$ as follows:

$$E \Downarrow_G v \quad \cong \quad [\emptyset \vdash E \rightsquigarrow f : T \quad \wedge \quad f \to^* v]$$

We use $E \Uparrow_G$ to denote that the GTLC term $E$ diverges.

Now we can define the semantics of our Rank-2 calculus. First we define how to translate a Rank-2 term $E$ to a GTLC term $\mathcal{T}(E)$. This will be relevant for theorem 4.4.2; see Figure 4.4. The translation scheme $\mathcal{T}$ replaces each intersection type with the precision-order greatest upper bound of the entries in the intersection. Additionally, $\mathcal{T}$ removes the label from each variable. Otherwise, $\mathcal{T}$ leaves the term unchanged.

Second, we define that a Rank-2 term $E$ evaluates to a cast-calculus value $v$, written $E \Downarrow v$, as follows.

$$E \Downarrow v \quad \cong \quad [\emptyset \vdash \mathcal{T}(E) \rightsquigarrow f : T \quad \wedge \quad f \to^* v]$$

Above we see how we first translate $E$ to the GTLC term $\mathcal{T}(E)$ and then proceed with the GTLC semantics of $\mathcal{T}(E)$. We use $E \Uparrow$ to denote that the Rank-2 term $E$ diverges.

We define $E \Downarrow_S v$ as follows:

$$E \Downarrow_S v \quad \cong \quad [\emptyset \vdash \mathcal{T}(E) \rightsquigarrow f : T \quad \wedge \quad f \rightarrow^* v]$$

## 4.4 Our Type System satisfies Standard Criteria for Gradual Types

We first prove that our type system satisfies three basic properties (Section 5.1), after which we prove that our type system satisfies the criteria in [SVC15a] (Sections 5.2–5.4).

### 4.4.1 Basic Properties

As foreshadowed in Section 4.2, our type system assigns every expression a unique type. Additionally, translation from Rank-2 to GTLC preserves typability and is monotonic.

**Theorem 4.4.1** (Unique Type)**.** $\forall \Gamma, E, T, T'$*, if* $\Gamma \vdash E : T$ *and* $\Gamma \vdash E : T'$*, then* $T = T'$*.*

See Appendix B.1 for the proof.

**Theorem 4.4.2** (Typability preservation)**.** $\forall \Gamma, E, T$ *if* $\Gamma \vdash E : T$ *then* $\mathcal{T}(\Gamma) \vdash_G \mathcal{T}(E) : \mathcal{T}(T)$*.*

*Proof.* By induction on $E$, using that deriving the greatest upper bound preserves typability. □

Notice that Theorem 4.4.2 uses the $\mathcal{T}$ function to map Rank-1 types to Rank-0 types. Recall that this is done by taking the greatest upper bound of the entries in the intersection.

**Theorem 4.4.3** (Monotonicity)**.** *If* $E_1 \sqsubseteq E_2$*, then* $\mathcal{T}(E_1) \sqsubseteq_G \mathcal{T}(E_2)$*.*

*Proof.* By induction on $E_1$. □

### 4.4.2 Gradual as a Superset of Static and Dynamic

Our gradual Rank-2 type system is a conservative extension of static Rank-2.

**Theorem 4.4.4** (Conservative Extension of Static Rank-2)**.** *For all static* $\Gamma, E$ *and* $T$*, we have:*

*1.* $\Gamma \vdash_S E : T$ *iff* $\Gamma \vdash E : T$*.*

*2.* $E \Downarrow_S v$ *iff* $E \Downarrow v$

See Appendix B.1 for the proof.

Our gradual Rank-2 type system is a conservative extension of the GTLC. Let $\text{GTLC}^0$ be the set of Rank-2 terms in which all the types are of Rank-0. When we apply $\mathcal{T}$ on terms in $\text{GTLC}^0$, the only effect is to remove the label from each variable. Thus, intuitively, $\mathcal{T}$ is the identity function on $\text{GTLC}^0$. The proof can be found in Appendix B.1.

**Theorem 4.4.5** (Conservative Extension of GTLC)**.** *For any* $\Gamma$ *that uses only Rank-0 types,* $E$ *in* $GTLC^0$*, and a Rank-0 type* $T$*, such that* $\Gamma$ *is defined only on free variables of* $E$*:*

*1.* $\Gamma \vdash E : T$ *iff* $\Gamma \vdash_G \mathcal{T}(E) : T$*.*

*2.* $E \Downarrow v$ *iff* $\mathcal{T}(E) \Downarrow_G v$

*Proof.* Siek et al. [SVC15a] showed that GTLC extends both the STLC and DTLC. So, given that Rank-2 is a conservative extension of GTLC, we have by transitivity that Rank-2 extends both the STLC and the DTLC. □

Following an idea of Siek and Taha [ST06], which is used also by Siek at al. [SVC15b], we encode the DTLC into the gradual Rank-2 calculus using the following mapping [.]:

$$
\begin{aligned}
[true] &= true \\
[false] &= false \\
[n] &= n \\
[x] &= x \\
[\lambda x.e] &= \lambda x : \texttt{Dyn}.[e] \\
[e_1\ e_2] &= ((\lambda x : \texttt{Dyn}.x)[e_1])[e_2]
\end{aligned}
$$

Let $\Downarrow_D$ denote evaluation of DTLC.

**Theorem 4.4.6** (Embedding of DTLC)**.** *Suppose that $E$ is a term of the DTLC.*

*1.* $\vdash [E] : \texttt{Dyn}$.

*2.* $E \Downarrow_D v$ *iff* $[E] \Downarrow v$.

*Proof.* Notice that Gradual Rank-2 conservatively extends GTLC by theorem 4.4.5, so since DTLC can be embedded in GTLC, then it can also be embedded in Gradual Rank-2. □

### 4.4.3 Soundness

We were unable to obtain a traditional soundness theorem while simultaneously satisfying our migration criteria as well as the static gradual guarantee. Instead, we satisfy a weaker theorem which we will state in this section. Before we state the theorems, let us state the background definition.

Following [SVC15b], we define subtyping for Rank-0 types, see Figure 4.5.

$$\texttt{Int} <: \texttt{Int} \quad \texttt{Bool} <: \texttt{Bool} \quad \texttt{Dyn} <: \texttt{Dyn}$$

$$\frac{\tau^0 <: \sigma^0}{\texttt{Dyn} <: \sigma^0} \qquad \frac{\sigma_0^0 <: \tau_1^0 \quad \tau_2^0 <: \sigma_2^0}{\tau_1^0 \to \tau_2^0 <: \sigma_0^0 \to \sigma_0^2}$$

**Figure 4.5:** Subtyping for Rank-0 types.

As we described before, our Type Safety theorem is weaker than the traditional soundness theorem. We weakened the theorem by applying the transformation function $\mathcal{T}$ to the Rank-2 type annotation $T$, which translates it to a Rank-0 type annotation. This transformation could potentially erase type information. The same applies to our Blame-Subtyping theorem. In summary, the theorems do not make guarantees about the original type of the program, but rather, about a well-defined, Rank-0 translation of this type. This yields a different theorem than the traditional soundness theorem, causing the system to be unsound with respect to the types in the source program. It is unclear weather our type system could satisfy the traditional type safety theorem while satisfying all of the remaining properties.

**Theorem 4.4.7** (Type Safety)**.** *If $\emptyset \vdash E : T$, then either $E \Downarrow v$ and $\emptyset \vdash v : \mathcal{T}(T)$ for some $v$, or $E \Downarrow blame_{\mathcal{T}(T)}\mathcal{L}$ for some $\mathcal{L}$, or $E \Uparrow$.*

*Proof.* Straightforward from the definition of $\mathcal{T}$, Theorem 4.4.2, and that GTLC itself satisfies type safety. □

**Theorem 4.4.8** (Blame-Subtyping)**.** *If $\emptyset \vdash \mathcal{T}(E) \rightsquigarrow f : T$ and $f$ contains a cast $f' : T_1 \Rightarrow^{\mathcal{L}} T_2$ and $T_1 <: T_2$ and $E \Downarrow blame_T\mathcal{L}'$, then $\mathcal{L} \neq \mathcal{L}'$*

*Proof.* Immediate from the definition of $\mathcal{T}$ and that GTLC itself satisfies blame-subtyping. □

### 4.4.4 The Gradual Guarantee

Our Rank-2 calculus satisfies the static gradual guarantee (item 1 below) and some items from the dynamic gradual guarantee (items 2–3 below), while items 4 and 5 are weaker varations of the dynamic gradual guarantee.

**Theorem 4.4.9** (Gradual Guarantee)**.** $\forall \Gamma, \Gamma', E, E', T$ *suppose* $\Gamma \vdash E : T$ *and* $\Gamma' \sqsubseteq \Gamma$ *and* $E' \sqsubseteq E$*.*

1. *For some* $T'$*, we have* $\Gamma' \vdash E' : T'$ *and* $T' \sqsubseteq T$*.*
2. *If* $E \Downarrow v$ *then* $E' \Downarrow v'$ *and* $v' \sqsubseteq v$
3. *If* $E \Uparrow$ *then* $E' \Uparrow$
4. *If* $E' \Downarrow v'$ *then* $E \Downarrow v$ *where* $v' \sqsubseteq v$ *or* $E \Downarrow blame_{\mathcal{T}(T)}\mathcal{L}$
5. *If* $E' \Uparrow$ *then* $E \Uparrow$ *or* $E \Downarrow blame_{\mathcal{T}(T)}\mathcal{L}$*.*

*Proof.* We prove item 1 in Appendix B.2, while items 2–5 are straightforward from the definition of $\mathcal{T}$, Theorem 4.4.3 (for items 2+4), and that GTLC itself satisfies the gradual guarantee. □

## 4.5 Migration

In this section, we show that the three migration problems found to be decidable for the GTLC are also decidable for our Rank-2 intersection type discipline. The key technical problem is to represent the migration space in a way that enables us to design algorithms for the migration problems. We consider a constraint based approach.

In Chapter 3, we showed that for the GTLC, the singleton problem, the top-choice problem, and the finiteness problem are all decidable, while the maximality problem is NP-hard. Let us consider each problem in turn and discuss what changes when we go from the GTLC to our Rank-2 intersection type discipline.

A program for which the migration space is a singleton in both the GTLC and our type system is $\lambda x : \texttt{Int}.x$. The reason is that no type is more precise than `Int`. In the GTLC, also the program $\lambda x : \texttt{Dyn}.x\ (succ\ x)$ has a singleton migration space. The reason is that $x$ cannot be a number and a function at the same time. However, in our type system, this is no longer true: we can migrate the program to $\lambda x : (\texttt{Int} \to \texttt{Int}) \wedge \texttt{Int}.x\ (succ\ x)$.

A program for which the migration space has a $\leq_\Gamma$-greatest element in the GTLC is $\lambda x : \texttt{Dyn} \to \texttt{Dyn}.(x\ 4) + (x\ True))$. Indeed, the top migration in the GTLC is $\lambda x : \texttt{Dyn} \to \texttt{Int}.(x\ 4) + (x\ True))$. This migration is $\leq_\Gamma$-maximal in our type system but not $\leq_\Gamma$-greatest. We note that once we have annotated $x$ with a type of the form $T^0 \to T^0$, we cannot migrate the annotation to a type of the form $(T^0 \wedge T^0) \to T^0$ because this would lead to a type outside rank 2. However, in our type system we can migrate the original program to $\lambda x : (\texttt{Int} \to \texttt{Int}) \wedge (\texttt{Bool} \to \texttt{Int}).(x\ 4) + (x\ True)$, which is $\leq_\Gamma$-maximal. A different example is $(\lambda x : \texttt{Dyn}.x)4$, which both in the GTLC and in our type system has the $\leq_\Gamma$-greatest migration $(\lambda x : \texttt{Int}.x)4$.

An example of a program with a finite migration space in the GTLC is $succ((\lambda y : \texttt{Dyn}.y)((\lambda x : \texttt{Dyn}.x)True))$. Note that since each variable has a single occurrence, we are restricted to use $T^0$ types in the annotations of $x$ and $y$. In both the GTLC and our type system, the migration space has three elements, which, aside from the original one, are $succ((\lambda y : \texttt{Int}.y)((\lambda x : \texttt{Dyn}.x)True))$ and $succ((\lambda y : \texttt{Bool}.y)((\lambda x : \texttt{Dyn}.x)True))$. However, finiteness in the GTLC does not imply finiteness in our type system. For example, $\lambda x : \texttt{Dyn}.x\ (succ\ x)$, which has a singleton migration space in GTLC, as we saw earlier, has an infinite migration space in our type system. The reason is that we can migrate the program to $\lambda x : (\texttt{Int} \to \sigma) \wedge \texttt{Int}.x\ (succ\ x)$ for any $\sigma$.

Notice that the previous example relies on that our language has a finite number of base types. This holds for many widely used languages with an infinitely countable set of types. For example, Java has eight built-in types: boolean, byte, char, short, int, long, float, and double, and we can compound them in systematic ways, allowing us to enumerate the space.

The program $\lambda x : \texttt{Dyn}.x\ x$ has no maximal migration in the GTLC, but it has a maximal migration in our type system, namely $\lambda x : (\texttt{Int} \to \texttt{Int}) \wedge \texttt{Int}.x\ x$. Indeed, it has infinitely many maximal migrations of the form $\lambda x : (\sigma' \to \sigma) \wedge \sigma'.x\ x$ where $\sigma$ and $\sigma'$ are $\sqsubseteq$-maximal $T^0$ types.

### 4.5.1 Constraints

For a given program, we will represent the migration space as the set of solutions to a constraint problem.

#### 4.5.1.1 Type variables.

Assume that $E$ has been $\alpha$-converted so that all bound variables are distinct from each other and distinct from the free variables. Let $X$ be the set of $\lambda$-variables $x$ occurring in $E$, and let $Y$ be a set of variables disjoint from $X$ consisting of a variable $[\![F]\!]$ for every occurrence of the subterm $F$ in $E$. Let $Z$ be a set of variables disjoint for $X$ and $Y$ consisting of a variable $\langle G \rangle$ for every occurrence of the subterm $(F\ G)$ in $E$. The notations $[\![F]\!]$ and $\langle G \rangle$ are ambiguous because there may be more than one occurrence of some subterm $F$ in $E$ or some subterm $G$ in $E$. However, it will always be clear from context which occurrence is meant.

A constraint is given by the grammar:

$$\begin{aligned} C \ ::= \ & C_1 \wedge C_2 \ \mid \ C_1 \vee C_2 \ \mid \ \tau \sqsubseteq (v_1^0 \wedge \ldots \wedge v_n^0) \ \mid \ v_1^0 \sim v_2^0 \ \mid \\ & v^0 = \tau^0 \ \mid \ v_1^0 = v_2^0 \ \mid \ v^0 = v_1^0 \to v_2^0 \ \mid \\ & v_1^2 = (v_1^0 \wedge \ldots \wedge v_n^0) \to v_2^2 \ \mid \ v^2 = (v_1^0 \wedge \ldots \wedge v_n^0) \to v^0 \end{aligned}$$

Let the set $\mathsf{proper}(T^2)$ be the subset of $T^2$ types that each contains at least one intersection type with at least two conjuncts. In the grammar above, $v^0, v_1^0, v_2^0, \ldots, v_n^0$ are type variables that range over $T^0$ types, while $v^2, v_1^2, v_2^2$ are type variables that range over $\mathsf{proper}(T^2)$ types.

Suppose $C$ is a constraint and $\varphi$ is a mapping that maps each type variable $v^0$ to a $T^0$ type and that maps each type variable $v^2$ to a $\mathsf{proper}(T^2)$ type. We define $\varphi \models C$ (pronounced: $\varphi$ solves $C$) by the recursive definition in Figure 4.6. We use $Sol(C)$ to denote the set of solutions of $C$. Let $Dom(\varphi)$ denote the domain of a mapping $\varphi$. We order mappings as follows:

$$\varphi \leq \varphi' \quad \Longleftrightarrow \quad Dom(\varphi) = Dom(\varphi') \ \wedge \ \forall v \in Dom(\varphi) : \varphi(v) \sqsubseteq \varphi'(v)$$

$$
\begin{array}{rcl}
\varphi \models C_1 \wedge C_2 & \iff & \varphi \vDash C_1 \wedge \varphi \vDash C_2 \\
\varphi \models C_1 \vee C_2 & \iff & \varphi \vDash C_1 \vee \varphi \vDash C_2 \\
\varphi \models \tau \sqsubseteq (v_1^0 \wedge \ldots \wedge v_n^0) & \iff & \tau \sqsubseteq (\varphi(v_1^0) \wedge \ldots \wedge \varphi(v_n^0)) \\
\varphi \models v_1^0 \sim v_2^0 & \iff & \varphi(v_1^0) \sim \varphi(v_2^0) \\
\varphi \models v^0 = \tau^0 & \iff & \varphi(v^0) = \tau^0 \\
\varphi \models v_1^0 = v_2^0 & \iff & \varphi(v_1^0) = \varphi(v_2^0) \\
\varphi \models v^0 = v_1^0 \to v_2^0 & \iff & \varphi(v^0) = \varphi(v_1^0) \to \varphi(v_2^0) \\
\varphi \models v_1^2 = (v_1^0 \wedge \ldots \wedge v_n^0) \to v_2^2 & \iff & \varphi(v_1^2) = (\varphi(v_1^0) \wedge \ldots \wedge \varphi(v_n^0)) \to \varphi(v_2^2) \\
\varphi \models v^2 = (v_1^0 \wedge \ldots \wedge v_n^0) \to v^0 & \iff & \varphi(v^2) = (\varphi(v_1^0) \wedge \ldots \wedge \varphi(v_n^0)) \to \varphi(v^0)
\end{array}
$$

**Figure 4.6:** Definition of $\varphi \models C$.

### 4.5.2 A Constraint-based Representation of the Migration Space

#### 4.5.2.1 The idea.

Figure 4.7 shows how we generate constraints for an expression. Each judgment is of the form $\Gamma \vdash E : i, t^2 | C$ where $\Gamma$ is a type environment, $E$ is an expression, $i \in \{0, 2\}$ is an input tag, $t^2$ is a skeleton, and $C$ is a constraint. Let us go over what each of those components mean. $\Gamma$ is the type environment under which our initial program $E$ type checks. We always assume a well-typed program. $i$ can be thought of as an input to the constraint generator and it stands for the migration space ranks. If our input tag is 0, our constraints will only describe the Rank-0 migration space. Otherwise, our constraints will describe the Rank-2 migration space, which also includes the Rank-0 migration space. In particular, if the input tag is 0 then we generate equivalent constraints to [MP19] because our calculus conservatively

extends GTLC. Therefore, we may capture the Rank-0 space by telling the generator to only generate constraints for Rank-0 migrations.

We require that $i$ is greater than or equal to the rank of the type of $E$ because a program that is strictly Rank-2 cannot have a Rank-0 migration. This follows directly from the definition of precision and the definition of a migration. Hence, the requirement ensures that all migrations are at least as precise as the input term. We can think of constraint generation as a function that takes a type environment $\Gamma$, an expression, and a tag as input and returns a skeleton and a constraint as output. The type environment is only for variables such as $+$ that are free in the top-level term. A skeleton describes an "upper bound" on the structure of a type. In particular, it tells us that when a migration has to have a Rank-2 type, we can determine exactly how many conjuncts there are and where those conjuncts are located in the type. This is useful for programs that can have both, a Rank-0 type and a Rank-2 type. For example, $\lambda x : \texttt{Dyn}.xx$ is such a program. In this case, we are saying that when $\lambda x : \texttt{Dyn}.xx$ has a Rank-2 type, the skeleton will be $2 \to 1$, which means we will have an arrow type with two conjuncts in the domain. This is what we mean by the "structure" of the type. The skeleton generated for a program will always be meaningful. In particular, we will never generate a strictly Rank-2 skeleton for a constant, because a constant will always have a simple type.

Each type variable in our constraints carries a tag representing a rank. For example, type variables for program variables are tagged with 0, in accordance with the *T-Var* rule. Similarly, type variables for arguments are also tagged with 0, in accordance with the *T-App* rule. For other expressions, we will tag a variable with the minimum rank possible. For example, if a variable ends up with the type `Int`, while that could be tagged with 0 or 2, we will tag that variable with 0. So variables tagged with tag 2 are strictly Rank-2 and not

Rank-0. Generally, our constraints consider all possibilities for different tags, as we will see later. We will distinguish occurrences of program variables via notation such as $x_0$ and $x_1$.

#### 4.5.2.2 Skeletons.

For the purpose of generating constraints that involve intersections, we need to know the number of conjuncts in each intersection. For this reason, we avoid use of Rank-1 variables and instead we introduce the concept of *skeletons*, see Figure 4.7. In the figure, 1 describes the structure of a Rank-0 type. A Rank-0 type can also have the structure $1 \to t^0$ where $t^0$ is the structure of a Rank-0 type. A $t^2$ can either be a $t^0$ or a type of the form $n \to t^2$. This mimics the structure of the grammar for types. In general, skeletons are similar to types except that they describe the structure of a type rather than an actual type. Note that the skeleton associated with a type variable describes its structure in case it is a higher rank. For example, a type variable for the expression $\lambda x.x + x$ has the skeleton $2 \to 1$. That is to say that if we want the Rank-2 solution, this is what the structure looks like. But we can also have the Rank-0 solution which must also be captured by the constraints. This is why our constraints have conjunctions of the same variable over different tags.

#### 4.5.2.3 Auxiliary functions.

We have auxiliary functions that determine the structure of the domain ($Dom$), Range ($Cod$) and Rank ($Tag$) of a given skeleton. For example, the domain and codomain of a Rank-0 type is also of Rank-0. For determining the rank of a variable, we compute the tag of the lowest rank: $Tag(n \to t) = 2$ when $n > 1$. When we see that $n > 1$, we know that this structure can only be of Rank-2. However, if we have $1 \to t$, we know that the result depends on $t$. If $t$ is Rank-0, the final result is Rank-0 and if it is Rank-2, the final result is of Rank-2.

#### 4.5.2.4 Shorthands.

We will use these shorthands. First, $v^0 \rhd v'^0 \to v''^0$ is a shorthand for

$$(v^0 = v'^0 \to v''^0) \vee ((v^0 = \texttt{Dyn}) \ \wedge \ (v'^0 = \texttt{Dyn}) \ \wedge \ (v''^0 = \texttt{Dyn}))$$

Additionally, we will use $v^2 \rhd (v_1^0 \wedge \ldots \wedge v_n^0) \to v''^2$ as a "shorthand" for $v^2 = (v_1^0 \wedge \ldots \wedge v_n^0) \to v''^2$, we will use $v^2 \rhd (v_1^0 \wedge \ldots \wedge v_n^0) \to v''^0$ as a "shorthand" for $v^2 = (v_1^0 \wedge \ldots \wedge v_n^0) \to v''^0$. Those last three shorthands are convenient for writing in a style where the superscript of the left-hand side of $\rhd$ is a meta-variable.

#### 4.5.2.5 The constraint generation rules.

Let us consider each rule in Figure 4.7. We use fresh type variables of the forms $x_l^0$, $x^0$, etc, and also $[\![E]\!]$ and $\langle E \rangle$, where $E$ is a term. The rule *S-True* assigns tag 0 to the type variable $[\![\text{True}]\!]$ and returns 1, which is a Rank-0 skeleton. The rules *S-False* and *S-Num* are similar.

We have two rules for variables. For a variable that is free in the top-level term, *S-Var-F* assigns the Rank-0 type in $\Gamma$, while for other variables *S-Var-B* allocates a new Rank-0 variable.

We have three rules for abstractions. Rule *S-Abs-0* handles the case where the input tag is 0. In this case, all tags must be 0, which is why we use 0 when generating constraints for the body $E$ of an abstraction $\lambda x : \tau.E$. For an abstraction $\lambda x : \tau.E$ with $n$ occurrences of $x$ in $E$, we must ensure that all elements of the type annotation are equal and have tag 0.

Rule *S-Abs-1* handles the case where a bound variable occurs $n$ times in the body of an abstraction. The skeleton we return must be of the form $n \to t$ because we have $n$ occurrences of $x$ in $E$. This will determine how many elements should be in the conjunction of the type

**Skeletons**:

$$\begin{array}{rcl} t^2 & ::= & n \to t^2 \mid t^0 \\ t^0 & ::= & 1 \mid 1 \to t^0 \\ n & ::= & n \in \mathbb{N} \end{array}$$

**Auxiliary functions**:

$$\begin{array}{rcl rcl rcl l} Dom(n \to t) & = & n & Cod(n \to t) & = & t & Tag(n \to t) & = & 2 \quad \text{if } n > 1 \\ Dom(1) & = & 1 & Cod(1) & = & 1 & Tag(1 \to t) & = & Tag(t) \\ & & & & & & Tag(1) & = & 0 \end{array}$$

**Constraints:**

$$\Gamma \vdash \texttt{true} : i, 1 \mid [\![True]\!]^0 = Bool \ \textit{(S-True)} \quad \Gamma \vdash \texttt{false} : i, 1 \mid [\![False]\!]^0 = Bool \ \textit{(S-False)}$$

$$\Gamma \vdash n : i, 1 \mid [\![n]\!]^0 = Int \ \textit{(S-Num)} \quad \Gamma \vdash x_0 : i, 1 \mid x_0^0 = \Gamma(x) \textit{(S-Var-F)}$$

$$\Gamma \vdash x_l : i, 1 \mid x_l^0 = [\![x_l]\!]^0 \textit{where } x \notin Dom(\Gamma) \ \textit{(S-Var-B)}$$

$$\frac{\Gamma \vdash E : 0, t \ \textit{ where } x \textit{ occurs in } E \ n \textit{ times and } n \geq 1 \mid C}{\Gamma \vdash (\lambda x : \tau^0.E) : 0, 1 | C \wedge \Big( [\![\lambda x : \tau^0.E]\!]^0 = x^0 \to [\![E]\!]^0 \wedge \tau^0 \sqsubseteq x^0 \wedge x^0 = x_1^0 \wedge x^0 = x_n^0 \Big)} \ \textit{(S-Abs-0)}$$

$$\frac{\begin{array}{c}\Gamma \vdash E : 2, t \ \textit{ where } x \textit{ occurs in } E \ n \textit{ times and } n \geq 1 \mid C \qquad j = Tag(t) \\ l = Tag(n \to t)\end{array}}{\begin{array}{l}\Gamma \vdash (\lambda x : \tau^1.E) : 2, n \to t \mid C \wedge \Big( \big( [\![\lambda x : \tau^1.E]\!]^l = (\bigwedge^n x_i^0) \to [\![E]\!]^j \vee \\ \qquad [\![\lambda x : \tau^1.E]\!]^l = (\bigwedge^n x_i^0) \to [\![E]\!]^0 \big) \wedge \\ \qquad \tau^1 \sqsubseteq (\bigwedge^n x_i^0) \vee \\ \qquad \big( ([\![\lambda x : \tau^1.E]\!]^j = x^0 \to [\![E]\!]^j \vee \\ [\![\lambda x : \tau^1.E]\!]^0 = x^0 \to [\![E]\!]^0) \wedge \tau^1 \sqsubseteq x^0 \big) \wedge x^0 = x_1^0 \wedge \ldots \wedge x^0 = x_n^0 \Big)\end{array}} \ \textit{(S-Abs-1)}$$

$$\frac{\Gamma \vdash E : i, t \ \ j = Tag(t) \ \textit{ where } x \textit{ does not occur in } E \mid C}{\begin{array}{l}\Gamma \vdash \lambda x : \tau^0.E : i, 1 \to t \mid C \wedge \Big( \big( [\![\lambda x : \tau^0.E]\!]^j = x^0 \to [\![E]\!]^j \vee \\ \qquad [\![\lambda x : \tau^0.E]\!]^0 = x^0 \to [\![E]\!]^0 \big) \wedge \tau^0 \sqsubseteq x^0 \Big)\end{array}} \ \textit{(S-Abs-2)}$$

$$\frac{\Gamma \vdash E_1 : 0, t_1 \mid C_1 \qquad \Gamma \vdash E_2 : 0, t_2 \mid C_2}{\Gamma \vdash E_1 \ E_2 : 0, 1 \mid C_1 \wedge C_2 \wedge [\![E_1]\!]^0 \rhd \langle E_2 \rangle^0 \to [\![E_1 \ E_2]\!]^0 \wedge \langle E_2 \rangle^0 \sim [\![E_2]\!]^0} \ \textit{(S-App-0)}$$

$$\frac{\begin{array}{c}\Gamma \vdash E_1 : 2, t_1 \mid C_1 \qquad \Gamma \vdash E_2 : 0, t_2 \mid C_2 \\ j = Tag(Cod(t_1)) \qquad n = Dom(t_1) \ \ l = Tag(t_1)\end{array}}{\begin{array}{l}\Gamma \vdash E_1 \ E_2 : 2, Cod(t_1) \mid C_1 \wedge C_2 \wedge \Big( \big( ([\![E_1]\!]^l \rhd \langle E_2 \rangle_1^0 \wedge \ldots \wedge \langle E_2 \rangle_n^0 \to [\![E_1 \ E_2]\!]^j \\ \qquad \vee [\![E_1]\!]^l \rhd \langle E_2 \rangle_1^0 \wedge \ldots \wedge \langle E_2 \rangle_n^0 \to [\![E_1 \ E_2]\!]^0) \\ \qquad \wedge (\bigwedge_{i \in \{1,\ldots,n\}} \langle E_2 \rangle_i^0 \sim [\![E_2]\!]^0) \big) \vee \\ \big( ([\![E_1]\!]^j \rhd \langle E_2 \rangle^0 \to [\![E_1 \ E_2]\!]^j \vee ([\![E_1]\!]^0 \rhd \langle E_2 \rangle^0 \to [\![E_1 \ E_2]\!]^0) \wedge \\ \qquad \langle E_2 \rangle^0 \sim [\![E_2]\!]^0) \big) \Big)\end{array}} \ \textit{(S-App-1)}$$

**Figure 4.7:** Constraint generation

of the abstraction. However, it does not imply that the abstraction can only get a Rank-2 type, which is why we interpret the skeleton as an upper bound for the rank. We must also consider the possibility of the abstraction having a Rank-0 type. Specifically, we have the disjunction ($[\![\lambda x : \tau.E]\!]^l = \bigwedge^n x_i^0 \to [\![E]\!]^j \vee [\![\lambda x : \tau.E]\!]^l = \bigwedge^n x_i^0 \to [\![E]\!]^0$) because for $[\![E]\!]$, the skeleton $t$ is an upper bound on the rank, which is why we tag $[\![E]\!]$ with $j$ and also 0. The same idea goes for the constraint ($[\![\lambda x : \tau.E]\!]^j = x^0 \to [\![E]\!]^j \vee [\![\lambda x : \tau.E]\!]^0 = x^0 \to [\![E]\!]^0$. Here, note that we tag $[\![E]\!]$ with 0 and since $x$ is tagged with 0 so $[\![\lambda x : \tau.E]\!]$ also gets the tag 0. We also explicitly determine the number of conjuncts in our Rank-2 type. Note that we may have duplicate constraints in case $j$ happens to be 0, but this will not affect the solvability or the time complexity of the solution procedure.

Rule *S-Abs-2* considers the possibility where a bound variable does not occur in the body of the abstraction. Then we know we should tag $x$ with 0 and we have two possibilities for $j$ since the skeleton $t$ of $E$ is an upper bound on the tag.

Rule *S-App-0* for an application $E_1\ E_2$ generates a constraint known from Chapter 3. The constraints generated here capture the Rank-0 migrations generated for applications while another rule generates constraints that capture the Rank-2 migrations. Rule *S-App-1* considers the number of conjunctions we need for $E_1$ using the skeleton given by $E_1$, but also considers the possibility that $E_1$ is of Rank-0. Notice that $j$ is the tag of the codomain of $t_1$, though this is only an upper bound, which is why we must not only tag $E_1\ E_2$ with $j$, but with 0 as well. Specifically, rule *S-App-1* has both ($[\![E_1]\!]^l \rhd \langle E_2\rangle_1^0 \wedge ... \wedge \langle E_2\rangle_n^0 \to [\![E_1\ E_2]\!]^j \vee [\![E_1]\!]^l \rhd \langle E_2\rangle_1^0 \wedge ... \wedge \langle E_2\rangle_n^0 \to [\![E_1\ E_2]\!]^0$) and ($[\![E_1]\!]^j \rhd \langle E_2\rangle^0 \to [\![E_1\ E_2]\!]^j \vee ([\![E_1]\!]^0 \rhd \langle E_2\rangle^0 \to [\![E_1\ E_2]\!]^0$). Again, $j$ may happen to be 0, resulting in a duplicate constraint.

Note that all rules ending in $-0$ are necessary because they allow us to capture the migrations that belong to the GTLC. Having tags on type variables that specify the rank

allows each type variable to represent a single expression which is a key to generalizing the approach in Chapter 3.

#### 4.5.2.6 Correctness.

Our constraint represents the migration space in the sense that the set of solutions of the constraint is order-isomorphic with the migration space.

**Definition 4.1.** Suppose that $\Gamma \vdash E : T$ and $\Gamma \vdash E : i, t \mid C$ for $i \in \{0, 2\}$ where the rank of $T$ is less than or equal to $i$. Then $Gen(E, \Gamma, i) = C$.

**Theorem 4.5.1.** $\forall E, \Gamma :$ *if* $FV(E) \subseteq Dom(\Gamma)$ *then* $(Mig_\Gamma(E), \sqsubseteq)$ *and* $(Sol(Gen(E, \Gamma, 2)), \leq)$ *are order-isomorphic.*

See Appendix B.3 for the proof.

### 4.5.3 Example of how Constraint Generation Works

Let us demonstrate how the constraint generation rules in Figure 4.7 work by applying them to the example $E_0 = (\lambda x : \texttt{Dyn}.(x_0\ 4) + (x_1\ True))(\lambda y : \texttt{Dyn}.5)$. Note that we use $x_0$ and $x_1$ to distinguish the two occurrences of $x$. We will use $\Gamma = [+ : \texttt{Int} \to \texttt{Int} \to \texttt{Int}]$ with the input tag 2, so we will derive $\Gamma \vdash E_0 : 2, 1 \mid C$, where $C$ is shown in Figure 4.8.

#### 4.5.3.1 Derivation.

We can use the rules in Figure 4.7 to produce the following derivation tree for $E_0$, omitting the constraints, and omitting a lookup in $\Gamma$ to get the type of $+$:

$$\dfrac{\dfrac{\dfrac{\Gamma \vdash x_0 : 0,1 \quad \Gamma \vdash 4 : 0,1}{\Gamma \vdash x_0\ 4 : 0,1}\ \textit{S-App-0} \quad \dfrac{\Gamma \vdash x_1 : 0,1 \quad \Gamma \vdash \mathit{True} : 0,1}{\Gamma \vdash x_1\ \mathit{True} : 0,1}\ \textit{S-App-0}}{\Gamma \vdash (x_0\ 4) + (x_1\ \mathit{True}) : 2,1}\ \textit{S-App-0}}{\Gamma \vdash \lambda x : \texttt{Dyn}.(x_0\ 4) + (x_1\ \mathit{True}) : 2, 2 \to 1}\ \textit{S-Abs-1}$$

Notice that $\lambda x : \texttt{Dyn}.(x_0\ 4) + (x_1\ \mathit{True})$ has the skeleton $2 \to 1$. The full derivation tree is then as follows:

$$\dfrac{\dfrac{\dots}{\Gamma \vdash \lambda x : \texttt{Dyn}.(x_0\ 4) + (x_1\ \mathit{True})) : 2, 2 \to 1}\ \textit{S-Abs-1} \quad \dfrac{\Gamma \vdash 5 : 0,1}{\Gamma \vdash \lambda y : \texttt{Dyn}.5 : 0,1}\ \textit{S-Abs-0}}{\Gamma \vdash (\lambda x : \texttt{Dyn}.(x_0\ 4) + (x_1\ \mathit{True}))(\lambda y : \texttt{Dyn}.5) : 2,1}\ \textit{S-App-1}$$

#### 4.5.3.2 Duplication.

In Figure 4.8, notice that the constraint has eight occurrences of $(C' \vee C')$ for different cases of $C'$, which is due to the simplicity of the example. For a particular case of this pattern, let us consider the last step of deriving $\Gamma \vdash E_0 : 2,1 \mid C$, which is done with the *S-App-1* rule. In the first two lines of the conclusion of the *S-App-1* rule, we have a disjunction of two matching constraints that here become:

$$\begin{aligned} [\![\lambda x : \texttt{Dyn}.(x_0\ 4) + (x_1\ \mathit{True})]\!]^2 &\rhd (v_1^0 \wedge v_2^0) \to [\![E_0]\!]^0 \vee \\ [\![\lambda x : \texttt{Dyn}.(x_0\ 4) + (x_1\ \mathit{True})]\!]^2 &\rhd (v_1^0 \wedge v_2^0) \to [\![E_0]\!]^0 \end{aligned}$$

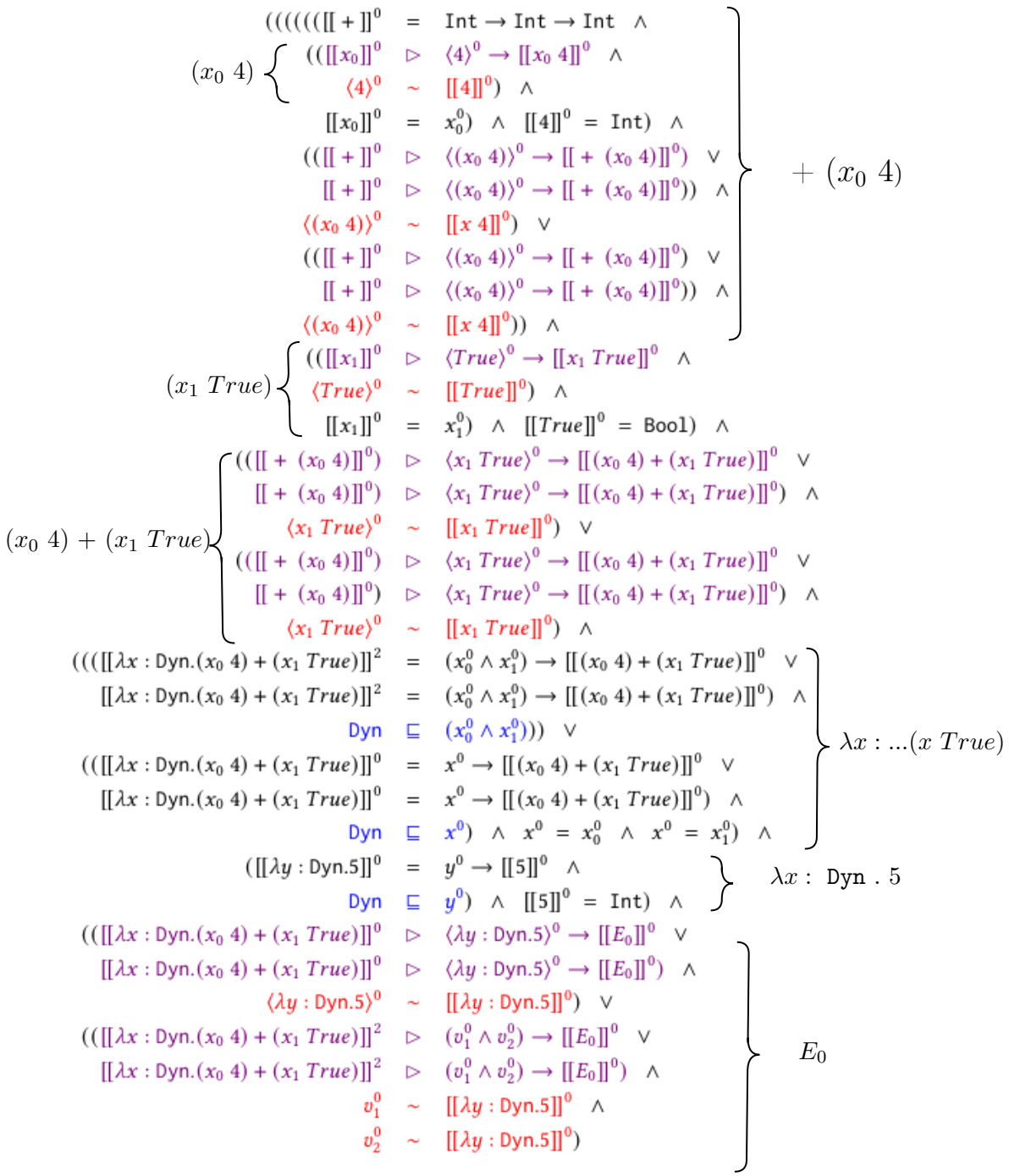


**Figure 4.8:** The constraint for $E_0 = (\lambda x : \texttt{Dyn}.(x_0\ 4) + (x_1\ True))(\lambda y : \texttt{Dyn}.5)$.

The tags on the left-hand sides are the same by definition, while the tags of $[\![E_0]\!]$ is $j$ in one entry and 0 in the other. However, here $j = 0$; why is that? To answer this question, we must consider one of the hypotheses of *S-App-1*, which here is: $\Gamma \vdash (\lambda x : \texttt{Dyn}.(x_0\ 4) + (x_1\ True)) : 2, t_1 \mid C_1$. Here, $t_1 = 2 \to 1$, so $j = Tag(Cod(t_1)) = Tag(Cod(2 \to 1)) = Tag(1) = 0$. In summary, in the *S-App-1* rule, the tag $j$ may be 0, which is already covered by a different constraint. By the way, notice how the skeleton $2 \to 1$ was the key to generating the right number of entries in each intersection.

Duplication can also happen with the rule *S-Abs-1*. The last rule used to derive $\Gamma \vdash (\lambda x : \texttt{Dyn}.(x_0\ 4) + (x_1\ True)) : 2, t_1 \mid C_1$ is *S-Abs-1*. In the first two lines of the conclusion of the *S-App-1* rule, a disjunction of two equality constraints that here become:

$$\begin{aligned} [\![\lambda x : \texttt{Dyn}.(x_0\ 4) + (x_1\ True)]\!]^2 &= (x_0^0 \wedge x_1^0) \to [\![(x_0\ 4) + (x_1\ True)]\!]^0 \quad \vee \\ [\![\lambda x : \texttt{Dyn}.(x_0\ 4) + (x_1\ True)]\!]^2 &= (x_0^0 \wedge x_1^0) \to [\![(x_0\ 4) + (x_1\ True)]\!]^0 \end{aligned}$$

The tags on the left-hand sides are the same by definition, while the tags of $[\![(x_0\ 4)+(x_1\ True)]\!]$ is $j$ in one entry and 0 in the other. Also in this case can we do a calculation that shows that $j = 0$.

Intuitively, the example program has no nested Rank-1 types, which leads to simple skeletons, like $2 \to 1$. Those simple skeletons, in turn, lead to constraints of the form $(C' \vee C')$.

#### 4.5.3.3 Different ranks.

The function $\lambda x : \texttt{Dyn}.(x_0\ 4) + (x_1\ True)$ might end up with a Rank-0 type or a Rank-2 type, and we need to be open to both possibilities. We achieve that in Figure 4.8 via a disjunction over two constraints, one for each possibility. Specifically, we have:

$$
\begin{array}{rcl}
(\llbracket \lambda x : \texttt{Dyn}.(x_0\ 4) + (x_1\ True) \rrbracket^2 & = & (x_0^0 \wedge x_1^0) \rightarrow \llbracket (x_0\ 4) + (x_1\ True) \rrbracket^0 \ \wedge \ \ldots\ ) \ \vee \\
(\llbracket \lambda x : \texttt{Dyn}.(x_0\ 4) + (x_1\ True) \rrbracket^0 & = & x^0 \rightarrow \llbracket (x_0\ 4) + (x_1\ True) \rrbracket^0 \ \wedge \ \ldots\ )
\end{array}
$$

In contrast, the type variables that represent the function $\lambda y : \texttt{Dyn}.5$, which are of the form $\llbracket \lambda y : \texttt{Dyn}.5 \rrbracket$, are always tagged with 0. This is because $\lambda y : \texttt{Dyn}.5$ is an argument in a function call and arguments must be rank 0.

#### 4.5.3.4 Consistency.

The *S-App-1* rule produces constraints that ensure consistency between the argument type and the type that the function expects. Like above, we need to be open to two possibilities: the function may end up with a type of rank 0 and or with a type of rank 2. For the function call $E_0$, this manifests itself in Figure 4.8 as the following disjunction:

$$
\begin{array}{rcl}
(\llbracket \lambda x : \texttt{Dyn}.(x_0\ 4) + (x_1\ True) \rrbracket^0 & \rhd & \langle \lambda y : \texttt{Dyn}.5 \rangle^0 \rightarrow \llbracket E_0 \rrbracket^0 \ \wedge \\
\langle \lambda y : \texttt{Dyn}.5 \rangle^0 & \sim & \llbracket \lambda y : \texttt{Dyn}.5 \rrbracket^0 ) \ \vee \\
(\llbracket \lambda x : \texttt{Dyn}.(x_0\ 4) + (x_1\ True) \rrbracket^2 & \rhd & (v_1^0 \wedge v_2^0) \rightarrow \llbracket E_0 \rrbracket^0 \ \wedge \\
v_1^0 & \sim & \llbracket \lambda y : \texttt{Dyn}.5 \rrbracket^0 \ \wedge \ v_2^0 \ \sim \ \llbracket \lambda y : \texttt{Dyn}.5 \rrbracket^0 )
\end{array}
$$

Notice that in the case of rank 2, the constraint ensures consistency with each of the entries of the intersection. Notice also that those consistency constraints are over Rank-0 types.

### 4.5.4 The Finiteness Problem

#### 4.5.4.1 Algorithm outline.

For solving the finiteness problem for a program $E$, we first generate a constraint and then we apply a decision procedure that is similar to the one in Chapter 3. Here we give an overview of the decision procedure; full details are in Appendix B.4.

The first step is to transform away all precision constraints. Constraints of the form $\texttt{Dyn} \sqsubseteq v$ are removed while the rest are simplified to be equalities over Rank-0 types. The next step is to transform the constraint into disjunctive normal form (DNF), that is, a disjunction of conjunctions. Each conjunction contains equality and consistency constraints. The use of DNF enables us to consider each clause of the DNF separately. The idea is that the set of solutions for the entire constraint is finite if and only if the set of solutions for each clause of the DNF is finite. The next step is to check each clause for finiteness, which we do with a procedure similar to the one in [MP19]. The only minor difference has to do with handling Rank-2 type variables, which turns out to be straightforward because they appear only in equations. The finiteness checker first solves the equations, then applies the most general unifier to the consistency constraints, and finally checks that each type variable is bounded by a $\sqsubseteq$-maximum type.

Recall that an instance of the Finiteness problem is a program.

**Theorem 4.5.2.** *We can solve the Finiteness problem in EXPTIME.*

See Appendix B.4 for the proof.

Our type system is a conservative extension of the GTLC, which means that a program that follows the GTLC syntax is well-typed in GTLC only if it can be typed in our type system with the same type. One consequence of that is that if the migration space of a program is finite in Rank-2, then it is also finite in GTLC.

**Theorem 4.5.3.** *If a program has a finite migration space in gradual Rank-2 then it has a finite migration space in GTLC.*

*Proof.* Straightforward. □

#### 4.5.4.2 Example.

Let us return to our running example $(\lambda x : \texttt{Dyn}.(x_0\ 4) + (x_1\ True))(\lambda y : \texttt{Dyn}.5)$ and check it for finiteness. First we generate the constraint in Figure 4.8, and then we notice that all three precision constraints are of the form $\texttt{Dyn} \sqsubseteq v^i$ and can be removed. Next, we convert the constraint to DNF and proceed to process each clause separately. For one of those clauses, once we solve the equalities and apply the most general unifier to the consistency constraints, we have:

$$\langle True \rangle^0 \sim \texttt{Int} \ \wedge \ [\![x_1\ True]\!]^0 \sim \texttt{Int} \ \wedge \ \langle True \rangle^0 \sim \texttt{Bool} \ \wedge \ \langle True \rangle^0 \sim y^0$$

In the first three entries of the conjunction, we have consistency constraints that each bounds a type variable. However, the fourth entry is a consistency constraint among two type variables and we see that no constraint bounds $y^0$. So, we conclude that the set of solutions for this constraint is infinite, hence the space of solutions for the entire constraint in Figure 4.8 is infinite.

### 4.5.5 The singleton problem

In Chapter 3, we solved the singleton problem for the GTLC via a *stepping function* $\mathcal{S}$; we will do something similar. Intuitively, $\mathcal{S}$ generates a set of types that are "one level above" a given type (and similarly for terms) in the precision relation. This works because the GTLC precision relation has finite intervals. Fortunately, our calculus has this property as well, which enables us to navigate a migration space effectively.

**Theorem 4.5.4** (Finite Intervals)**.** $\forall E_l, E_u : \{E \mid E_l \sqsubseteq E \sqsubseteq E_u\}$ *is finite.*

Straightforward from the definition of $\sqsubseteq$ and the stepping function $\mathcal{S}$.

To generalize the stepping function from Chapter 3, we must define it on $T^0$ types and $T^1$ types, and then extend it to terms. In each case, the stepping function takes a minimal step along the precision relation. Since annotations have two different ranks, we have one function for each rank. For Rank-0 we define the stepping function in the same way as [MP19]. For Rank-1, additionally we must handle intersections. We observe that the only non-trivial minimal step from `Dyn` to a strict Rank-1 type is $\texttt{Dyn} \wedge \texttt{Dyn}$. Any other type will be at least two steps above `Dyn`. We must also consider the Rank-0 types that are one level above `Dyn` and we union the cases. Stepping for an intersection type is done in a similar way to stepping from an arrow type. We then extend the stepping function to terms in a straightforward way. Here follows the detailed definition.

We begin by defining $\mathcal{S}^0$ on $T^0$ types.

$$
\begin{aligned}
\mathcal{S}^0(\texttt{Bool}) &= \emptyset \\
\mathcal{S}^0(\texttt{Int}) &= \emptyset \\
\mathcal{S}^0(\texttt{Dyn}) &= \{\ \texttt{Bool},\ \texttt{Int},\ \texttt{Dyn} \to \texttt{Dyn}\ \} \\
\mathcal{S}^0(\tau_1 \to \tau_2) &= \bigcup_{\tau_1' \in \mathcal{S}^0(\tau_1)} \{\ \tau_1' \to \tau_2\ \} \cup \bigcup_{\tau_2' \in \mathcal{S}^0(\tau_2)} \{\ \tau_1 \to \tau_2'\ \}
\end{aligned}
$$

Next, we define $\mathcal{S}^1$ over $T^1$. This definition extends $\mathcal{S}^0$ to $T^1$ types. Specifically, when called on $\texttt{Dyn}$, it combines the result of $\mathcal{S}^0$ with $\texttt{Dyn} \wedge \texttt{Dyn}$. $\mathcal{S}^1$ handles the $\wedge$ constructor in the same way that $\mathcal{S}^0$ did on $\to$ constructors.

$$
\begin{aligned}
\mathcal{S}^1(\sigma) &= \mathcal{S}^0(\sigma) \quad \text{where } \sigma \in T^0 \wedge \sigma \neq \texttt{Dyn} \\
\mathcal{S}^1(\texttt{Dyn}) &= \mathcal{S}^0(\texttt{Dyn}) \cup \{\texttt{Dyn} \wedge \texttt{Dyn}\} \\
\mathcal{S}^1(\tau_1 \wedge \tau_2) &= \bigcup_{\tau_1' \in \mathcal{S}^1(\tau_1)} \{\ \tau_1' \wedge \tau_2\ \} \cup \bigcup_{\tau_2' \in \mathcal{S}^1(\tau_2)} \{\ \tau_1 \wedge \tau_2'\ \}
\end{aligned}
$$

We now define $\mathcal{S}$ on terms in the same way as the GTLC. The difference is that since annotations are $T^1$ types, we call $\mathcal{S}^1$ instead of $\mathcal{S}^0$.

$$
\begin{aligned}
\mathcal{S}(n) &= \mathcal{S}(\texttt{true}) = \mathcal{S}(\texttt{false}) = \mathcal{S}(x) = \emptyset \\
\mathcal{S}(\lambda x : T.E) &= \bigcup_{T' \in \mathcal{S}^1(T)} \{\ \lambda x : T'.E\ \} \cup \bigcup_{E' \in \mathcal{S}(E)} \{\ \lambda x : T.E'\ \} \\
\mathcal{S}(E_1 E_2) &= \bigcup_{E_1' \in \mathcal{S}(E_1)} \{\ E_1' E_2\ \} \cup \bigcup_{E_2' \in \mathcal{S}(E_2)} \{\ E_1 E_2'\ \}
\end{aligned}
$$

Our singleton checker works in the same way as the one in Chapter 3 for the GTLC. Specifically, to check that the migration space for a term $E$ is a singleton, it calculates $\mathcal{S}(E)$ and verifies that every element of $E$ doesn't type check.

**Theorem 4.5.5.** *We can solve the singleton problem in polynomial time.*

*Proof.* $\mathcal{S}$ generates the next level of the migration space in polynomial time because each annotation leads to a finite number of branches. □

### 4.5.6 The Top-choice problem

The Top Choice problem deals with whether a migration space contains a greatest element. We can solve the Top Choice problem using the finiteness checker and the stepping function. This relies on the following property, which is also true of the GTLC.

**Theorem 4.5.6.** *If* $\texttt{Mig}_\Gamma(E)$ *has a greatest element, then* $\texttt{Mig}_\Gamma(E)$ *is finite.*

*Proof.* Suppose $Mig_\Gamma(E)$ has a greatest element $E_g$, which means that any migration $E'$ must satisfy $E \sqsubseteq E' \sqsubseteq E_g$. Thus, $Mig_\Gamma(E) \subseteq \{\ E' \mid E \sqsubseteq E' \sqsubseteq E_g\ \}$. We have from Theorem 4.5.4 that $\{\ E' \mid E \sqsubseteq E' \sqsubseteq E_g\ \}$ is finite so also $Mig_\Gamma(E)$ is finite. □

Our top-choice checker works in the same way as the one in Chapter 3 for the GTLC. Specifically, we first check that the migration space is finite, and then we use the stepping function to search for a greatest element.

**Theorem 4.5.7.** *We can solve the top-choice problem in EXPTIME.*

*Proof.* The execution time is dominated by the finiteness checker, which runs EXPTIME (Theorem 4.5.2). □

### 4.5.7 The Maximality problem

In Chapter 3, we showed that for the GTLC, the Maximality problem is NP-hard while they gave no decidability result. For our calculus of Rank-2 intersection types, we have neither a lower bound nor a decidability result for the Maximality problem. In particular, the proof of NP-hardness in Chapter 3 doesn't work because it relies on that all uses of a variable have the same type. In contrast, our calculus enables different uses of a variable to have different types. For example, consider the following unsatisfiable Boolean formula $F_3$.

$$\begin{aligned} F_3 \quad = \quad & (x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee \bar{x}_3) \wedge (x_1 \vee \bar{x}_2 \vee x_3) \wedge (x_1 \vee \bar{x}_2 \vee \bar{x}_3) \wedge \\ & (\bar{x}_1 \vee x_2 \vee x_3) \wedge (\bar{x}_1 \vee x_2 \vee \bar{x}_3) \wedge (\bar{x}_1 \vee \bar{x}_2 \vee x_3) \wedge (\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3) \end{aligned}$$

The proof of NP-hardness of Maximality in Chapter 3 encodes $F_3$ as a program with no maximal migration in GTLC. However, for that same program we used our implementation to find a maximal migration with Rank-2 types. We view the Maximality problem as an interesting and practically relevant topic for future work.

In the absence of a decidability result for Maximality, we note that our stepping function enables us to navigate a migration space effectively. This provides a semi-algorithm for finding maximal migrations in a breadth-first-search manner, if such migrations exist. Indeed, in our experiments, we found maximal migrations for all programs in the benchmark suite in Chapter 3, including those for which their semi-algorithm for GTLC timed out. Those benchmarks are exclusively untyped terms, that is, all variables are annotated with `Dyn`. This opens the question of whether every untyped term has a maximal migration in our type system. We leave this to future work.

## 4.6 Experimental Results

Our experimental evaluation answers two questions about using our type system for type migration. The questions and our claims are as follows.

1. Does our type system lead to different migrations than GTLC? *Yes, we get different migrations for some microbenchmarks.*

2. Does our type system lead to more maximal migrations compared to GTLC? *Yes, for our two largest microbenchmarks, we found maximal migrations, while none are known in GTLC.*

In the remainder of this section, we show how our evaluation supports our claims.

We implemented our algorithms in 1,400 lines of Haskell. We will present experimental results from running our implementation on the benchmark suite in Chapter 3 plus our running example. Figure 4.9 lists the benchmarks, except *selfInterpreter*, which is the lambda-term

$$Y[\lambda e.\lambda m.m(\lambda x.x)(\lambda mn.(em)(en))(\lambda m.\lambda v.e(mv))]$$

which is a self-interpreter for pure lambda-calculus [Mog92]. It uses the abbreviation $Y = \lambda h.(\lambda x.h(xx))(\lambda x.h(xx))$.

For all benchmarks, we use $\Gamma = [\texttt{succ} : \texttt{int} \to \texttt{int}\ ,\ + : \texttt{int} \to \texttt{int} \to \texttt{int}]$.

We compare the results with those in Chapter 3 to illustrate how the type systems provide different maximal migrations and migration spaces.

We ran the finiteness check, the singleton check, the top-choice check and the maximality check on our benchmark suite.

### 4.6.1 The Singleton Check

For GTLC, a single benchmark, $\lambda x.x(\texttt{succ}(x))$, in the benchmark suite has a singleton migration space. However, none of the benchmarks has a singleton migration space in our type system. In particular, we can migrate $\lambda x.x(\texttt{succ}(x))$ to $\lambda x : (\texttt{Int} \to \texttt{Int}) \wedge \texttt{Int}.x\ (succ\ x)$. As we described with our running example, these annotations describe the underlying program, and it is up to the programmer to decide if this is the program they intended to write.

### 4.6.2 The Top-Choice Check

For GTLC, each of four benchmarks has a migration space with a greatest element. However, in our type system, only one of those has a migration space with a greatest element. For example, $\lambda x : \texttt{Dyn} \to \texttt{Dyn}.(x\ 4) + (x\ True))$ has a top migration in the GTLC, namely $\lambda x : \texttt{Dyn} \to \texttt{Int}.(x\ 4) + (x\ True))$. This migration is $\leq_{\Gamma}$-maximal in our type system but not $\leq_{\Gamma}$-greatest, as we saw earlier. This indicates that we have expanded the migration space with more informative types.

### 4.6.3 The Finiteness Check

We ran experiments on the finiteness check for the benchmark suite in Figure 4.9 and the results confirm that if a benchmark has a finite migration space in our type system, then it has a finite migration space in GTLC (Theorem 4.5.3). The results also show that the converse is false, as evidenced by three benchmarks.

### 4.6.4 The Search for Maximality

Figure 4.9 shows results from running the maximality search. (In Figure 4.9 two of the programs take two lines, which we indicate with horizontal lines.)

We were able to find maximal migrations for all the benchmarks using our tool. Larger benchmarks like *selfInterpreter* required some user interaction, where we found more precise migrations, picked some picked a migration that we believe could lead to a maximal migration then ran the tool again on the new program until reaching a maximal migration.

In some situations, the maximal migration in our type system is more precise than that in GTLC, like in the case of the first benchmark. In some other situations, the maximal migration in GTLC was also maximal in Rank-2, but Rank-2 gave other migrations as well. This was the case for our second and third benchmarks.

In the case of $\lambda x.xx$, the program has no maximal migration in the GTLC, but has a maximal migration in our type system. For each of two other benchmarks, no maximal migration is known in the GTLC, but we found one for our type system. Notice that $[(\lambda x.\lambda y.y(xI)(xK))\Delta]'$ uses `Dyn` twice, while $[\mathit{selfInterpreter}]'$ uses `Dyn` four times. Thus, here we see nontrivial uses of advanced gradual types and we are able to make sense of them. For example, for $[(\lambda x.\lambda y.y(xI)(xK))\Delta]'$, the type of $x$ reflects that we apply $x$ to $I$ and $K$ and the type of $\lambda d.dd$ gives the best type that fits the type of $x$. For *selfInterpreter*, we found a useful gradual intersection type for the $Y$ combinator.

As for the performance, 10 of our benchmarks run between 0.00 seconds to 2.11 seconds, with 8 benchmarks running under 0.25 seconds. We have manually found maximal migrations for the three larger benchmarks as we described above. For the second and third benchmarks of our table, we have first annotated the program with $\texttt{Dyn} \wedge \texttt{Dyn}$ because our tool currently returns the first maximal migration available, which would be a GTLC maximal migration

in this case. Since we are able to navigate the migration space though, several other ways can be explored to return specific migrations. Finally, for the last example, after annotating the program with $\texttt{Dyn} \wedge \texttt{Dyn}$, the closest maximal migration that is returned by the tool is different than the one in the table, but the migration in the table can also be found by probing the tool to return the list of migrations at a given level.

## 4.7 Flexible Rank-2: Adding Associativity and Commutativity

Automatic program migration is a non-trivial problem which required many language restrictions, resulting in loss of flexibility in the source language. In this section, we propose a methodology for introducing flexibility to our language without losing important migratory properties. We propose a separation of concerns between migration of programs and flexibility of the source language. Specifically, we design a flexible user-facing language containing additional language features that allow the ease of use. A program in this language will then be translated to a corresponding program in the core language where migration can occur. Migrated programs will then be translated back to migrations in the flexible language.

We demonstrate this concept by extending our initial type system with commutativity and associativity in a system we refer to as Flexible Rank-2, which subsumes gradual Rank-2 (Theorem 4.7.3). Commutativity adds flexibility to our type system as it allows the programmer to change the code without necessarily changing the corresponding type annotations. The rest of the section is organized as follows. In subsection 4.7.1 we introduce commutativity and associativity to our type system by defining a notion of equivalence and introducing it to our typing rules. We demonstrate the added flexibility of the system with an example. In subsection 4.7.2, we introduce the notion of migration in Flexible Rank-2 and show that every well-typed Flexible Rank-2 program can be mapped to an equivalent well-typed Gradual Rank-2 program (Theorem 4.7.1). We then demonstrate that every migration in gradual Rank-2 can be mapped to an equivalent migration in Flexible Rank-2 (Theorem 4.7.2). Finally, in subsection 4.7.3, we show that all gradual properties that hold for gradual Rank-2 also hold for Flexible Rank-2.

### 4.7.1 Flexible Rank-2

Figure 4.10 contains our typing rules. Notice that all typing rules are similar to those of Gradual Rank-2 except for the *(T-Abs-1-f)* rule. In an abstraction $\lambda x : \sigma.E$, this rule allows the current type environment containing $x : \sigma_0^0 \wedge ... \wedge \sigma_n^0$ to match the correct variable occurrence in an abstraction with the correct conjunct, while admitting any equivalent annotation $\sigma \cong \sigma_0^0 \wedge ... \wedge \sigma_n^0$, which allows us to admit an abstraction of the form $\lambda x : \sigma.E$, instead of limiting well-typedness to $\lambda x : \sigma_0^0 \wedge ... \wedge \sigma_n^0.E$. Equivalence is defined to admit commutativity and associativity. The definition of consistency does not vary under equivalence because consistency is defined over Rank-0 types, while the equivalence relation is concerned with Rank-1 and Rank-2 types. The definition of matching also does not vary under equivalence as it deals with arrow types. Note that Flexible Rank-2 subsumes Gradual Rank-2 because the rule *(T-Abs-1)*, uses equality instead of equivalence between the type annotation and the type in the type environment. Indeed, the type system in Figure 4.10 subsumes the core Rank-2. We state this formally in theorem 4.7.3.

Definition 4.2 formally defines an equivalence relation that admits commutativity and associativity which is then used in the typing rules. For the remaining theory, we will also need to extend the definition to terms (Definition 4.3), as we will see later in the section.

**Definition 4.2** (Type Equivalence)**.**

$$\tau \wedge \sigma \cong \sigma \wedge \tau$$

$$\tau_1 \wedge (\tau_2 \wedge \tau_3) \cong (\tau_1 \wedge \tau_2) \wedge \tau_3$$

**Definition 4.3** (Term Equivalence)**.**

$$E \cong E$$

$$\lambda x : \tau.E \cong \lambda x : \tau'.E' \iff E \cong E' \wedge \tau \cong \tau'$$

$$E_1\ E_2 \cong E_1'\ E_2' \iff E_1 \cong E_1' \wedge E_2 \cong E_2'$$

Let us present an example of how Flexible Rank-2 provides better usability. The following two programs are valid under our new language.

$$(\lambda x : (\texttt{Int} \to \texttt{Int}) \wedge (\texttt{Bool} \to \texttt{Int}).(x\ 4) + (x\ True))(\lambda y : \texttt{Dyn}.5) \tag{4.4}$$

$$(\lambda x : (\texttt{Bool} \to \texttt{Int}) \wedge (\texttt{Int} \to \texttt{Int}).(x\ 4) + (x\ True))(\lambda y : \texttt{Dyn}.5) \tag{4.5}$$

This implies that if the programmer modifies the body of the function to be $(x\ True)+(x\ 4)$, the programmer would not have to change the type annotation.

Notice that in 4.4, the derivation tree would look the same as that of our core language. in 4.5, the rule *(T-Abs-1-f)* will account for commutativity and associativity. We can see this by considering the following sub-derivation:

$$\frac{\begin{array}{c}\Gamma, (x : (\texttt{Int} \to \texttt{Int}) \wedge (\texttt{Bool} \to \texttt{Int})) \\ \vdash_f (x\ 4) + (x\ True) : \texttt{Int} \\ \sigma \cong (\texttt{Int} \to \texttt{Int}) \wedge (\texttt{Bool} \to \texttt{Int})\end{array}}{\Gamma \vdash_f \lambda x : (\texttt{Bool} \to \texttt{Int}) \wedge (\texttt{Int} \to \texttt{Int}).(x\ 4) + (x\ True) : \sigma \to \texttt{Int}}$$

In the sub-derivation, the type environment carries the correct order of conjuncts, which is $(\texttt{Int} \to \texttt{Int}) \wedge (\texttt{Bool} \to \texttt{Int})$. The equivalence relation admits $(\texttt{Bool} \to \texttt{Int}) \wedge (\texttt{Int} \to \texttt{Int})$ as a valid type annotation.

### 4.7.2 Migration Properties

In this section, our goal is to show that a separation of concerns between migration and flexibility is possible. We demonstrate this by proving that a well-typed program in Flexible Rank-2 can be mapped to an equivalant program in Gradual Rank-2. We also show that if a program is migrated in Gradual Rank-2, then there exists an equivalent migration in Flexible Rank-2.

Let us start with background definitions. Definition 4.4 extends Type Precision to Flexible Type Precision by relaxing the original definition. It states that there must exist a re-ordering of the types involved in the relation such that precision holds. Definition 4.5 extends this notion to terms in a similar fashion. With precision defined, we can extend the definition of migration to Flexible Rank-2 (definition 4.6) by replacing term precision with Flexible term precision in the original definition (definition **??**).

**Definition 4.4** (Flexible Type Precision)**.** $\tau \lesssim \sigma$ iff $\exists \tau' : \tau \cong \tau' \wedge \tau' \sqsubseteq \sigma$

**Definition 4.5** (Flexible Term Precision)**.** $E \lesssim E'$ iff $\exists E'' : E \cong E'' \wedge E'' \sqsubseteq E'$

**Definition 4.6** (Flexible Migration)**.** $E'$ is a flexible $\Gamma$-migration of $E$ (written $E \leq_{\Gamma_f} E'$) iff $(E \lesssim E' \ \wedge \ \exists T' : \Gamma \vdash_f E' : T')$.

Theorem 4.7.1 states that if we can type check a program in Flexible Rank-2, then we can type check an equivalent program in Gradual Rank-2 type system. The proof can be found in Appendix B.6.

**Theorem 4.7.1.** *If* $\Gamma \vdash_f E : T$ *then* $\exists T', E'$ *such that* $T' \cong T, E' \cong E$ *and* $\Gamma \vdash E' : T'$

Theorem 4.7.2 states that if we can migrate a program in our original type system, then we can migrate a program under equivalence in our flexible type system. The proof can be found in Appendix B.6.

**Theorem 4.7.2.** *Suppose* $\Gamma \vdash_f E : T$*. Then for* $E_s \cong E$ *with* $\Gamma \vdash E_s : T_s$*, if* $E_s \leq_\Gamma E'_s$*, then there exists an* $E'$ *such that* $E \leq_{\Gamma_f} E'$ *with* $E' \cong E'_s$*.*

Theorem 4.7.3 also clearly holds, and states that a term in Gradual Rank-2 is also well-typed under our language extension.

**Theorem 4.7.3** (Flexible Rank-2 subsumes Gradual Rank-2)**.** *If* $\Gamma \vdash E : T$ *then* $\Gamma \vdash_f E : T$

### 4.7.3 Gradual Properties

We will now restate the gradual criteria that we have proven for our core Rank-2 type system and prove them for our extension.

First, the definition of $\mathcal{T}$ from figure 4.4 still applies to our language because $\mathcal{T}$ does not differ on commutativity or associativity.

We define that $E \Downarrow_f v$ as follows:

$$E \Downarrow_f v \;\; \cong \;\; [\emptyset \vdash \mathcal{T}(E) \rightsquigarrow f : T \;\; \wedge \;\; f \rightarrow^* v]$$

Finally, we define $E \Downarrow_S v$ as follows:

$$E \Downarrow_S v \;\; \cong \;\; [\emptyset \vdash \mathcal{T}(E) \rightsquigarrow f : T \;\; \wedge \;\; f \rightarrow^* v]$$

**Theorem 4.7.4** (Conservative Extension of Static Rank-2)**.** *For all static* $\Gamma, E$ *and* $T$*, we have:*

*1.* $\Gamma \vdash_S E : T$ *iff* $\Gamma \vdash_f E : T$*.*

*2.* $E \Downarrow_S v$ *iff* $E \Downarrow_f v$

*Proof.* Part 1 is immediate from the fact that Flexible Rank-2 is a conservative extension to Rigid Rank-2 and Rigid Rank-2 is a conservative extension to Static Rank-2. Part 2 is straightforward by considering the definitions of $\Downarrow_f$ and $\Downarrow_s$. □

We will now prove the gradual guarantee. We require an auxiliary theorem which we prove in appendix .

**Theorem 4.7.5.** *Let* $\Gamma \vdash E_s : T_s$ *and* $\Gamma \vdash_f E : T$ *and* $E_s \cong E$ *and* $T_s \cong T$*. Then for every* $E' \lesssim E$*, there exists some* $E'_s \cong E'$ *with* $E'_s \sqsubseteq E_s$*, where* $\forall \Gamma'$*, if* $\Gamma' \vdash E'_s : T'_s$*, then* $\Gamma' \vdash_f E' : T'$ *where* $T' \lesssim T$*,* $T'_s \sqsubseteq T_s$*,* $\Gamma' \sqsubseteq \Gamma$ *and* $T' \cong T'_s$*.*

**Theorem 4.7.6** (Gradual Guarantee)**.** $\forall \Gamma, \Gamma', E, E', T$ *suppose* $\Gamma \vdash_f E : T$ *and* $\Gamma' \sqsubseteq \Gamma$ *and* $E' \lesssim E$*.*

*1. For some* $T'$*, we have* $\Gamma' \vdash_f E' : T'$ *and* $T' \lesssim T$*.*

*2. If* $E \Downarrow_f v$ *then* $E' \Downarrow_f v'$ *and* $v' \lesssim v$

*3. If* $E \Uparrow_f$ *then* $E' \Uparrow_f$

*4. If* $E' \Downarrow_f v'$ *then* $E \Downarrow_f v$ *where* $v' \lesssim v$ *or* $E \Downarrow blame_{\mathcal{T}(T)}\mathcal{L}$

*5. If* $E' \Uparrow_f$ *then* $E \Uparrow_f$ *or* $E \Downarrow_f blame_{\mathcal{T}(T)}\mathcal{L}$.

*Proof.* For item 1, note that if $\Gamma \vdash_f E : T$ then for some $E_s \cong E$ and $T_s \cong T$, $\Gamma \vdash E_s : T_s$, by theorem 4.7.1.

Given that $\Gamma' \sqsubseteq \Gamma$, then by theorem 4.7.6, we have that since $\Gamma \vdash E_s : T_s$, then for all $E'_s \sqsubseteq E_s$, $\Gamma' \vdash E'_s : T'_s$ for some $T'_s \sqsubseteq T_s$.

Note that since $E_s \cong E$, for every $E' \lesssim E$, there exists some $E'_s \cong E'$ with $E'_s \sqsubseteq E_s$. So from the above, we have $\Gamma' \vdash E'_s : T'_s$, for some $T'_s \sqsubseteq T_s$. From theorem 4.7.5, $\Gamma' \vdash_f E' : T'$ for some $T' \lesssim T$.

Items 2-5 are immediate from the definition of $\Downarrow_f$ and $\Uparrow_f$ and theorem 4.7.6.

**Theorem 4.7.7** (Embedding of DTLC)**.** *Suppose that* $E$ *is a term of the DTLC.*

*1.* $\vdash [E] : \texttt{Dyn}$.

*2.* $E \Downarrow_D v$ *iff* $[E] \Downarrow_f v$.

*Proof.* Follows immediately from the definition of $\Downarrow_f$ and theorem 4.4.6. □

**Theorem 4.7.8** (Type Safety)**.** *If* $\emptyset \vdash_f E : T$*, then either* $E \Downarrow_f v$ *and* $\emptyset \vdash_f v : \mathcal{T}(T)$ *for some* $v$*, or* $E \Downarrow blame_{\mathcal{T}(T)}\mathcal{L}$ *for some* $\mathcal{L}$*, or* $E \Uparrow_f$.

*Proof.* The proof is straightforward from the definition of $\mathcal{T}$, which preserves equality under $\cong$, Theorem 4.4.2 and that GTLC itself satisfies type safety. □

**Theorem 4.7.9** (Blame-Subtyping)**.** *If* $\emptyset \vdash \mathcal{T}(E) \rightsquigarrow f : T$ *and* $f$ *contains a cast* $f' : T_1 \Rightarrow^{\mathcal{L}} T_2$ *and* $T_1 <: T_2$ *and* $E \Downarrow_f blame_T\mathcal{L}'$*, then* $\mathcal{L} \neq \mathcal{L}'$

*Proof.* Immediate from the definition of $\mathcal{T}$ which preserves equality under $\cong$ and that GTLC itself satisfies blame-subtyping. □

□

## 4.8 Related work

We will discuss related papers, as well as the language Typed Racket.

### 4.8.1 Three related papers.

| Property | Castagna and Lanvin 2017 | Castagna et al. 2019 | Ângelo and Florido 2019 | Ours |
|---|---|---|---|---|
| Extension of a static type system | ? | ✓ | ? | ✓ |
| Extension of a dynamic type system | ✓ | ✓ | ? | ✓ |
| Type safety | ✓ | ✓ | ? | ? |
| Blame-subtyping | ? | ✓ | ? | ? |
| Static gradual guarantee | ? | ✓ | ? | ✓ |
| Dynamic gradual guarantee | ? | ? | ? | ? |
| The Singleton problem is decidable | ? | ? | ? | ✓ |
| The Top Choice problem is decidable | ? | ? | ? | ✓ |
| The Finiteness problem is decidable | ? | ? | ? | ✓ |
| The Maximality problem is decidable | ? | ? | ? | ? |

**Figure 4.11:** Comparison with closely related work.

In Figure 4.11 we summarize our comparison with three related papers by Castagne et al. [CL17], by Castagna et al. [CLP19], and by Angelo and Florido [AF19]. Figure 4.11 begins with Siek et al.'s criteria from 2015 and then lists the four migration problems defined in Chapter 3.

Castagna et al. [CL17] deal with full-fledged set-theoretic types, which makes the decidability problems a major open challenge. Their types satisfy the natural distribution laws. Section 4 of their paper states that types are unique, Theorem 3 in their paper says

that compilation is type preserving, and Theorem 2 states type safety. However, in section 5 the paper mentions that it is unknown whether it satisfies the static gradual guarantee, and in section 6 the paper says that a blame theorem is "not worth proving" for their calculus. The paper has no details about whether the type system is a conservative extension of a static type system or of the GTLC, and whether cast insertion is monotonic. Finally, the decidability problems remain open for this system.

Castagna et al. [CLP19] deals with gradual set-theoretic types that, like the types in [CL17], satisfy natural distribution laws. The type system also has subtyping, regularity, and contractivity. This is unlike our system where we do not consider subtyping, and where our types are not commutative or idempotent. Their system is also unrestricted in the number of conjuncts in an intersection type, while we limit the number of types in an intersection type by the number of variable occurrences. Their type system has no rank restriction on argument types, while ours does. We type check each argument once to be able to have a unique type for each expression. In contrast, their type system does not support unique types, which we can see from the Materialize rule in Figure 1 of their paper. We had private communication with one of the authors and learned that their system is a conservative extension of its static counterpart since the only difference between the static and gradual rules is their Materialize rule. Additionally, their Materialize rule also gives that their system satisfies that the static gradual guarantee. We also learned that it is still an open problem whether cast insertion is type preserving and monotonic. Theorem 4.11 in their paper states type safety, while Corollary 4.12 in their paper states a blame theorem that is closely related to blame-subtyping. Finally, the decidability problems remain open for this system.

The type systems by Castagna et al. [CL17] and by Castagna et al. [CLP19] are both more expressive than ours. The restrictions we imposed on our type system allow it to satisfy some of Siek et al's criteria [SVC15a] while enabling algorithmic support for type migration.

Angelo and Florido introduced a gradual Rank-2 intersection type system. In their type system, type intersection is idempotent and the typing rules for abstraction and application that are entirely different from ours. They do not consider Siek et al's criteria [SVC15a] . Their focus is on type inference but with the limitation that type inference cannot introduce intersection types: it only infers simple types. Thus, type migration like we showed for our running example is out of scope for the type inference algorithm by Angelo and Florido [AF19].

### 4.8.2 Typed Racket.

Typed Racket supports gradual intersection and union types. Both intersection types and union types are commutative and idempotent, and intersection types have no restriction on the rank. Typed Racket imposed a restriction that all top-level constructors in an intersection have to be disjoint. This restriction is called tidiness and was initially introduced by Wright [WC97]. However, this restriction has since been lifted. It remains an open question weather better migration tool support can be provided for gradual languages with that restriction.

Typed Racket uses *macro* gradual typing which means that we can annotate with types only at the module level, unlike our system which uses micro gradual typing, where we can annotate programs at a finer granularity. Accordingly, we will divide our running example into two modules. For comparison with our system, we consider following version of our running example:

```
1 #lang typed/racket
2  (: f : (Intersection (Number -> Number) (Boolean -> Number)) -> Number)
3  (define (f x)
4    (+ (x 4) (x true)))
5
6  (: g (Intersection (Number -> Number) (Boolean -> Number)))
7  (define (g x) 5)
8
9  (f g)
```

In the above code, we have two functions. The first one in line 3 defines the function `f` and the one in line 6 defines the function `g`. In line 9, `f` is called with `g` as an argument. The above code uses a intersection types which are similar to the ones in our system. Typed Racket does not support dynamic enforcement of intersection types.

### 4.8.3 Migrating Gradual Types

In this section, we discuss the work by Campora et al. [CCE18] and its relation to our work. One of the goals in both works is to find maximal migrations. The starting calculi in their work and ours are different. We consider Gradual Rank-2, while Campora et al. [CCE18] considers the ITGL, which is a language with a principle type inference algorithm. In contrast, Gradual Rank-2 does not have such an algorithm. The definitions of maximal migrations are different across both works. Our definition of maximality is based on the precision ordering, where a maximal migration must be the most precise migration, while in their work, a migration is maximal if no more `Dyn` can be eliminated. This leads to a key difference between their work and ours, which is that Campora et al. [CCE18] work with finite migration spaces only, which is why their definition of maximality is possible. In contrast, we consider the entire migration space of a program, which can be infinite. This yields scenarios where a program cannot be assigned a maximal migration because it does not exist. Indeed, in Chapter 3, we showed that this problem is NP-hard.

### 4.8.4 TypeWhich

In this subsection we discuss how Phipps-Costin et al. [PAG21]'s work would fit into the overall picture for our language.

Phipps-Costin et al. [PAG21] developed a theory and a tool called TypeWhich which follow a constraint-based approach for finding migrations based on several criterion. Their constraints can be expressed in Z3. We will discuss how might their approach be extended to our work. TypeWhich reasons about GTLC programs. It takes a GTLC program and migrates it, taking into account GTLC runtime semantics. Since our system is an extension of GTLC and also compiles to GTLC, we can adapt the same constraints for our reasoning

about runtime for our Rank-2 migrations. The main difference is that our migrations can also have Rank-2 annotations instead of only GTLC annotations, so we would have to add constraints which generate Rank-2 annotations.

We conjecture that this extension is possible. As we have seen from our constraints, which have a solution space that is order-isomorphic to the migration space, our constraints can be rewritten as equality constraints. We have seen examples of this when solving the finiteness problem. Provided that we know the number of conjuncts in an intersection, which we know by looking at the number of occurrences of bound variables, we can express our constraints in Z3. We conjecture that we can add a constraint which relates a GTLC migration to a Rank-2 migration by considering a type variable for a Rank-2 migration $v$ and a GTLC migration $v'$. The constraints would be of the form $v = v_1 \wedge ... \wedge v_n$ and $v' = v_1 \sqcap ... \sqcap v_n$. If we give our constraints, along with the constraints from TypeWhich, along with a constraint relating a GTLC migration to a Rank-2 migration to Z3, we believe that we could get Rank-2 migrations, which satisfy the GTLC runtime semantics, as outlined by Phipps-Costin et al. [PAG21].

In summary, we conjecture that we can extend TypeWhich's solution to accommodate Rank-2 annotations because our system compiles programs into GTLC, and that the remaining step is to encode our Rank-2 typing rules into Z3, with a constraint relating a GTLC migration to a Rank-2 migration, allowing Z3 to find a satisfying assignment. We will leave this to future work.

As it currently stands, Phipps-Costin et al.'s work is fundamentally different than ours in that their work considers context based migrations, while we only consider whole programs.

## 4.9 Conclusion

The gradual Rank-2 intersection type discipline is a powerful extension of the GTLC that satisfies key criteria for gradually typed languages. Our experiments show that Rank-2 types lead to more maximal migrations than GTLC. The decidability of three migration problems gives hope for scalable tool support for migration to Rank-2 intersection types.

**Limitations** In this work, we don't consider natural extensions of intersection types, such as union types [BDD95] and type-test operations [PAG21].

**future work** We leave to future work to consider those extensions as well as try to lift some of the restrictions on intersection types while achieving our goal. In particular, adding union types and type-test operations appear to be natural next steps for future work. We hope our results can serve as a foundation for designing languages with both gradual intersection types and support for type migration.

| Maximal GTLC | Maximal Rank-2 |
|---|---|
| $\lambda x : \texttt{Dyn}.x(\texttt{succ}(x))$ | $\lambda x : (\texttt{Int} \to \texttt{Int}) \wedge \texttt{Int}.x(\texttt{succ}(x))$ |
| $\lambda x : \texttt{Dyn} \to \texttt{Int}.x(\texttt{succ}(x(\texttt{true})))$ | $\lambda x : (\texttt{Int} \to \texttt{Int}) \wedge (\texttt{Bool} \to \texttt{Int}).x(\texttt{succ}(x(\texttt{true})))$ |
| $\lambda x : \texttt{Dyn} \to \texttt{Int}. + (x\ 4)(x\ \texttt{true})$ | $\lambda x : (\texttt{Int} \to \texttt{Int}) \wedge (\texttt{Bool} \to \texttt{Int}). + (x\ 4)(x\ \texttt{true})$ |
| $(\lambda x : \texttt{Int}.x)4$ | $(\lambda x : \texttt{Int}.x)4$ |
| $\texttt{succ}((\lambda y : \texttt{Dyn}.y)((\lambda x : \texttt{Int}.x)\texttt{true}))$ | $\texttt{succ}((\lambda y : \texttt{Dyn}.y)((\lambda x : \texttt{Int}.x)\texttt{true}))$ |
| $\lambda x : \texttt{Int}.x$ | $\lambda x : \texttt{Int}.x$ |
| $\lambda x : \texttt{Int}.\lambda y : \texttt{Int} \to \texttt{Int} \to \texttt{Int}.yxx$ | $\lambda x : \texttt{Int}.\lambda y : \texttt{Int} \to \texttt{Int} \to \texttt{Int}.yxx$ |
| $\lambda x : \texttt{Dyn}.(\lambda y : \texttt{Int}.x)xx$ | $\lambda x : \texttt{Dyn}.(\lambda y : \texttt{Int}.x)xx$ |
| $\lambda x : \texttt{Int}.(\lambda f : \texttt{Dyn}.$ $(\lambda x : \texttt{Dyn}.\lambda y : \texttt{Int}.x)f(fx))(\lambda z : \texttt{Int}.1)$ | $\lambda x : \texttt{Int}.(\lambda f : \texttt{Dyn} \wedge (\texttt{Int} \to \texttt{Int}).$ $(\lambda x : \texttt{Int}.\lambda y : \texttt{Int}.x)f(fx))(\lambda z : \texttt{Int}.1)$ |
| No maximal migration | $\lambda x : (\texttt{Int} \to \texttt{Int}) \wedge \texttt{Int}.xx$ |
| *unknown* | $[(\lambda x.\lambda y.y(xI)(xK))\Delta]'$ |
| *unknown* | $[\mathit{selfInterpreter}]'$ |
| $(\lambda x : \texttt{Dyn} \to \texttt{Int}.$ $(x\ 4) + (x\ True))\ (\lambda y : \texttt{Int}.5)$ | $(\lambda x : (\texttt{Int} \to \texttt{Int}) \wedge (\texttt{Bool} \to \texttt{Int}).$ $(x\ 4) + (x\ True))\ (\lambda y : \texttt{Dyn}.5)$ |

where

$$
\begin{aligned}
[(\lambda x.\lambda y.y(xI)(xK))\Delta]' \;=\;& (\lambda x : ((\texttt{Int} \to \texttt{Int}) \to \texttt{Int}) \wedge ((\texttt{Int} \to \texttt{Int} \to \texttt{Int}) \to \texttt{Int}).\\
& \lambda y : \texttt{Int} \to \texttt{Int} \to \texttt{Int}.\\
& y\ (x(\lambda a : \texttt{Int}.a))(x(\lambda b : \texttt{Int}.\lambda c : \texttt{Int}.b)))\\
& \quad (\lambda d : \texttt{Dyn} \to \texttt{Dyn}.dd)
\end{aligned}
$$

$$
\begin{aligned}
[\mathit{selfInterpreter}]' \;=\;& Y'[\lambda e : \texttt{Dyn}.\lambda m : \texttt{Dyn}.\\
& \quad m(\lambda x : \texttt{Int}.x)\\
& \quad\quad (\lambda m : \texttt{Int}.\lambda n : \texttt{Int}.(em)(en))\\
& \quad\quad (\lambda m : (\texttt{Int} \to \texttt{Int}).\lambda v : \texttt{Int}.e(mv))]\\
Y' \;=\;& \lambda h : (((\texttt{Int} \to \texttt{Int}) \to (\texttt{Int} \to \texttt{Int})) \wedge (\texttt{Int} \to \texttt{Dyn})).\\
& \quad (\lambda x : (((\texttt{Int} \to \texttt{Int}) \to (\texttt{Int} \to \texttt{Int})) \wedge (\texttt{Int} \to \texttt{Int})).h(xx))\\
& \quad (\lambda x : \texttt{Dyn}.h(xx))
\end{aligned}
$$

**Figure 4.9:** Maximal migrations.

Typing Rules

$\Gamma \vdash_f E : T^2$.

$\Gamma \vdash_f \texttt{true} : \texttt{Bool}$ *(T-True)* $\quad$ $\Gamma \vdash_f \texttt{false} : \texttt{Bool}$ *(T-False)* $\quad$ $\Gamma \vdash_f n : \texttt{Int}$ *(T-Num)*

$$\frac{x : \bigwedge_{i \in I} \tau_i^0 \in \Gamma \quad l \in I}{\Gamma \vdash_f x_l : \tau_l^0} \; (T\text{-}Var)$$

$$\frac{\begin{array}{c}\Gamma, (x : \sigma_0^0 \wedge \ldots \wedge \sigma_n^0) \vdash_f E : \tau \\ \textit{where } \forall i \in 0..n, x_i \textit{ occurs in } E \\ \textit{and } \sigma_0^0 = \ldots = \sigma_n^0 = \sigma^0\end{array}}{\Gamma \vdash_f (\lambda x : \sigma^0.E) : \sigma^0 \to \tau} \; \textit{(T-Abs-0)}$$

$$\frac{\begin{array}{c}\Gamma, (x : \sigma_0^0 \wedge \ldots \wedge \sigma_n^0) \vdash_f E : \tau \\ \textit{where } \forall i \in 0..n, x_i \textit{ occurs in } E \\ \textit{and } \sigma \cong \sigma_0^0 \wedge \ldots \wedge \sigma_n^0\end{array}}{\Gamma \vdash_f (\lambda x : \sigma.E) : \sigma \to \tau} \; \textit{(T-Abs-1-f)}$$

$$\frac{\Gamma \vdash_f E : \tau \;\; \textit{where } x_0 \textit{ does not occur in } E}{\Gamma \vdash_f (\lambda x : \tau^0.E) : \tau^0 \to \tau} \; \textit{(T-Abs-2)}$$

$$\frac{\begin{array}{cc}\Gamma \vdash_f E_1 : \sigma_1 & \Gamma \vdash_f E_2 : \sigma^0 \\ \sigma_1 \rhd ((\bigwedge_{i \in I} \tau_i^0) \to \sigma) & \forall i \leq I : \sigma^0 \sim \tau_i^0\end{array}}{\Gamma \vdash_f E_1 \; E_2 : \sigma} \; \textit{(T-App)}$$

Consistency:

$\sigma^0 \sim \texttt{Dyn}$ *(C-Dyn1)* $\quad$ $\texttt{Dyn} \sim \sigma^0$ *(C-Dyn2)* $\quad$ $\texttt{Bool} \sim \texttt{Bool}$ *(C-Bool)*

$\texttt{Int} \sim \texttt{Int}$ *(C-Int)*

$$\frac{\sigma^0 \sim \sigma'^0 \quad \tau^0 \sim \tau'^0}{(\sigma^0 \to \tau^0) \sim (\sigma'^0 \to \tau'^0)} \; \textit{(C-Arrow)}$$

Matching:

$(\sigma^1 \to \tau^2) \rhd (\sigma^1 \to \tau^2)$ *(M-Arrow)* $\quad$ $\texttt{Dyn} \rhd (\texttt{Dyn} \to \texttt{Dyn})$ *(M-Dyn)*

**Figure 4.10:** The Flexible Gradual Rank-2 intersection-typed $\lambda$-calculus: Typing Rules

# CHAPTER 5

## Generalizing Shape Reasoning with Gradual Types

Current machine learning libraries heavily rely on tensors because tensors provide an alternative to programming with scalars and nested loops. Tensors improve the process of program development and understanding; however, they obscure shape information in the program, which can cause shape errors that are difficult to detect. Shape analysis is beneficial beyond preventing shape errors. It can provide the compiler with information so it can optimize its resources. Shape analysis can also benefit various program transformation tools, which are common in machine learning, so that they can make valid program transformations. Because there are various contexts in which shape analysis is desirable, different works approach shape analysis from different angles. These works currently lack an underlying theory that unifies their approaches.

In tensor programs, expressing shapes through type annotations allows for effective shape-checking and reasoning. While shape reasoning is valuable, manually annotating programs with shapes is impractical and burdensome. However, gradual types offer a way to incrementally introduce type annotations into programs. Our research focuses on automatic type migration, which allows us to automatically annotate programs with shapes.

In this chapter, we develop a comprehensive gradual typing theory to reason about tensor shapes. We show that by asking different questions about the migration space of a gradually typed program, we can generalize shape analysis. We demonstrate this by solving three key

problems on shapes: (1) how to find a static migration, (2) how to find a migration that satisfies an arithmetic constraint, and (3) how to eliminate branches that depend on program shapes for an infinite class of inputs. In doing so, we develop a novel tool to address the first two problems. For the third problem. there are currently two PyTorch tools that aim to eliminate branches. They do so by eliminating it for just a single input. We show that our tool is the first to eliminate branches for an infinite class of inputs. Notice that these problems are related to the problems discussed in Chapter 2. However, our modified questions are relevant and useful for machine learning models. All programs we looked at are able to be fully statically typed unless they had a bug, hence question 1. Developers would like to query a tool for particular migrations based on some arithmetic constraints, hence question 2. Finally, most models contain no control-flow and one of the goals is to eliminate control-flow from the ones that have it, hence question 3.

## 5.1 Introduction

Multidimensional data structures are a common abstraction in modern machine learning frameworks such as PyTorch [PGC17], TensorFlow [AAB16], and JAX [BFH18]. A significant portion of programs written using these frameworks involve transformations on tensors. Tensors in this setting are $n$-dimensional arrays. A tensor is characterized by its *rank* and *shape*. The *rank* is the number of dimensions. For example, a matrix is two-dimensional; hence it is a rank-2 tensor. The *shape* captures the lengths of all axes of the tensor. For example, in a $2 \times 3$ matrix, the length of the first axis is 2 and the length of the second axis is 3; hence its shape is $(2, 3)$.

Programming with abstractions like tensors provides the programmer with high level and easy to understand constructs. The alternative approach is to program with scalars and nested

loops [PS21]. This approach makes programs harder to visualize and understand. While tensors bring ease of use to a programming language, their shapes are hard to track. Modern machine learning frameworks support a plethora of operations on tensors, with complex shape rules. PyTorch and TensorFlow for example support a mechanism called broadcasting which enables tensors which have different shapes and even dimensions to be compatible such that arithmetic operations could be performed on them. This involves modifying the tensors involved in broadcasting. However, modern frameworks only modify the results for optimally. Addition, for example, supports a mechanism called *broadcasting*, which allows us to add tensors of different shapes, provided that they satisfy a set of rules. Reshapes and convolutions give rise to complex shape rules with non-trivial arithmetic. Complex shape rules make shapes hard to determine in programs, because shape information rarely explicitly appears in them. As a result, shape errors occur frequently [ZCC18].

Undetected shape errors only appear at runtime, which is undesirable because we would only know about the error when the wrong operation is finally invoked on concrete runtime values. Tensor computations are costly and a program may take a long time to run before finally crashing with an error. In fact, shape errors may not even appear at runtime because running the program with an input might fail to catch the error because the error might only manifest for specific input shapes.

The ability to reason about shapes is useful in various contexts in the machine learning area. It can prevent programmers from making mistakes. It can also help compilers optimize for computational resources and program transformation tools to make valid program transformations. Users often add asserts or comments to help them reason about shapes. These tasks have a high cognitive load on users, especially when they are dealing with complex tensor operations. Shape asserts present even further challenges; they can manifest in the form of branches on program shapes. We observed this pattern on various transformer

benchmarks [Wol00]. Thus, in that pattern, the result of a branch depends on the shape of the program input, so the branch result can vary over different inputs. In machine learning programs, branches can be undesirable because they limit the back-ends a program can be run on [RDH21]. In practice, various tools handle this challenge in different ways. Some tools reject such programs entirely while other tools run the program on a single input to eliminate branches. Running a program on a single input means that branch elimination is correct for just one input, which is an unsatisfactory solution.

Aiming to prevent the need for ad-hoc shape asserts, entire systems have been build to detect shape errors such as [PS21] and [SLZ18]. However, these systems are too specific. They lack a general theoretical foundation that enables their solution to be adapted to a variety of contexts, including incorporating their logic into compilers and program transformation tools.

A fundamental approach towards shape analysis is building a type system that supports shapes and using it to reason about them. In that approach, shapes are type annotations. Traditionally, types have been used to solve similar problems in the area of programming languages [SSM19, SMS18]. We can consider a static type system with tensor shapes [RLW18]. A static type system has two limitations. First, a static type system may need to be elaborate in order to capture the complexities of machine learning programs, which are typically written in permissive languages such as Python. As a result, refinement or polymorphic types may be needed. Second, a static type system has a high barrier of entry because it requires the user to come up with non-trivial type annotations in advance. Third, many machine learning programs are in Python, so they are usually only partially typed. Therefore, fully typed programs are not readily available, which prevents this approach from being backwards-compatible.

A common way to circumvent the requirement of having a fully typed programs is to use gradual types. In a gradually typed system, type annotations are not needed for the program to compile. However, for a gradually typed system to be more widely usable, it should enable fundamental yet practical tool support. Previous work such as [HKS22] designed a gradually typed system for shapes. But the system is so powerful that practical, elaborate tool support can be hard to obtain. *We believe that the key to shape analysis with gradual types is to balance between (1) the expressiveness of a gradually typed system and (2) the ease of tool support in that system.*

We show that gradual types can help us tackle shape-related problems in a fundamental and unified way. We introduce a gradual typing system that reasons about shapes and enables tool support. We obtain a system that enables tool support via a syntactic interpretation of gradual types. We then define the migration space of a gradually typed program according to that interpretation. The concept of a migration space has been formally defined in Chapter 3. Let us revisit the intuition for it. Suppose we start with an untyped program. In a gradually typed language, there can be various but valid ways to annotate that program. Some of those possibilities include partially annotating the program with static types, which yields a *gradual migration*. Other possibilities involve annotating the entire program with static types, which yields a *static migration*. The *migration space* of a gradually typed program is the set of migrations that represent all correct ways of typing our initial program. We demonstrate that capturing the migration space enables various ways of reasoning about the shapes in a gradually typed tensor program.

We distill shape analysis challenges into three key problems that we can ask of every gradually typed tensor program to improve existing tool support, and we introduce a general theory to solve all of them:

Q(1) *Static migration:* Is the migration space non-empty? If so, does the migration space contain a static migration and if so, how do we find one?

Q(2) *Migration under arithmetic constraints*: Given an arithmetic constraint on a dimension, is there a migration that satisfies it and if so, how do we find one?

Q(3) *Branch elimination*: Can we prove that branch elimination is valid for an infinite set of inputs, not just for a single input? If so, how do we represent that set of inputs?

**Our results and contributions.** We choose PyTorch as a setting for our evaluation and build a tool, but our work is more generally applicable. First, we demonstrate how our algorithms work for only Q(1) and Q(2). PyTorch does not currently have any comparable tools for those problems. To address Q(3), we incorporate our reasoning into two existing PyTorch tools that aim to eliminate branches from PyTorch programs. After augmenting both tools with our logic, we are able to improve the performance and accuracy of both tools as we will describe below. Our contributions can be summarized as follows:

1. A Gradually Typed Tensor Calculus that satisfies standard static gradual criteria [SVC15a]
2. A formal characterization of the migration space via constraints
3. Algorithms for each of our three problems
4. A demonstration of how our algorithms work for Q(1) and Q(2) on four benchmarks
5. For Q(3), a comparison on six benchmarks, against `HFTracer` [Wol00], a PyTorch tool. `HFTracer` eliminates all branches on one input, while we eliminate all branches on infinite classes of inputs; we use constraints to represent infinite classes of inputs

6. For Q(3), a comparison on five benchmarks against `TorchDynamo` [Ans22], a PyTorch tool. `TorchDynamo` eliminates 0% of the branches in these benchmarks, while we eliminate branches by 40% to 100% on infinite classes of inputs

## 5.2 Three Migration Problems

### 5.2.1 Our type system, informally

In this section, we introduce our type system informally. We discuss the formal details in the next section. A tensor type in our system is of the form $\texttt{TensorType}(d_1, ..., d_n)$ where $d_1, ..., d_n$ are dimensions. Every gradually typed system has a type `Dyn`, which represents the absence of static type information. In our system, `Dyn` can appear as a dimension, in which case the dimension is unknown. `Dyn` can also appear as a tensor annotation, in which case even the rank of the tensor is unknown.

As discussed in the previous chapters, in gradual types, a precision relation refers to the replacement of some of the occurrences of `Dyn` with static types. `Dyn` is the least precise type because it contains no type information. $\texttt{TensorType}(1, 2, 3)$ and $\texttt{TensorType}(1, 2)$ are unrelated by the precision relation because we cannot go from one type to another by replacing `Dyn` occurrences with more informative types, while $\texttt{TensorType}(\texttt{Dyn}, 2)$ is less precise than $\texttt{TensorType}(1, 2)$ because we can replace the `Dyn` in $\texttt{TensorType}(\texttt{Dyn}, 2)$ with 1 to get $\texttt{TensorType}(1, 2)$. This relation extends to programs. Program $A$ is less precise than program $B$ if we can replace some occurrences of `Dyn` in program $A$ to get to program $B$. Intuitively, program $B$ is more static than program $A$. Precision gives rise to the *migration space*, as shown in the previous chapters. Given a well-typed program, its migration space is the set of more precise programs that are well-typed.

The migration space captures the different ways of correctly annotating a gradually typed program. As such, we can capture certain parts of the space by imposing appropriate constraints on it. With that in mind, let us look at examples of how reasoning about the migration space is beneficial for solving key problems about the shapes in a gradually typed program. Specifically, in Section 5.2.2, we will see two examples about Q(1) and Q(2) respectively, and in Section 5.2.3, we will see an example about Q(3).

### 5.2.2 Examples of static migrations and migration under arithmetic constraints

**Static migration** Consider Listing 5.1 which has a clear bug. Let us understand what the bug is.

**Listing 5.1:** Ill-typed convolution

```
10 class ConvExample(torch.nn.Module):
11     def __init__(self):
12         super(BasicBlock, self).__init__()
13         self.conv1 = torch.nn.Conv2d(in_channels=2, ..)
14         self.conv2 = torch.nn.Conv2d(in_channels=4, ..)
15
16     def forward(self, x: TensorType([Dyn, Dyn])):
17         self.conv1(x)
18         return self.conv2(x)
```

In line 7, `x` is annotated with `TensorType([Dyn, Dyn])`. This is a typical gradual typing annotation which indicates that `x` is a rank-2 tensor. The annotation does not specify what the dimensions are. In line 8, we are applying a convolution to `x`. According to PyTorch's documentation, for the convolution to succeed, `x` cannot be rank-2. Thus, the bug in this program stems from a wrong type annotation. The migration space of this program can easily

inform us that the program is ill-typed, because the space will be empty. The reason for that is that the migration space of a well-typed program should contain at least one element, which is the program itself. A tool that can reason about the migration space can easily catch this bug in a single step.

Let us fix this bug by replacing the wrong type annotation with a correct one. In Listing 5.2, we change `x`'s annotation from a rank-2 annotation to a rank-4 annotation: `TensorType([Dyn, Dyn, Dyn, Dyn]`, which is correct. This program compiles, but it contains a more subtle bug. Let us look closely at the code to understand why that is the case.

In line 4, we are initializing a field, `self.conv1`, representing a convolution, `torch.nn.Conv2d`, which takes various parameters. The parameter that's relevant to our point is called `in_channels` and it is set to 2. In line 5, we are initializing another field, `self.conv2`, but this time, we set the `in_channels` to 4. In line 7, we have a function that takes a variable `x` and calls both convolutions on it in lines 8 and 9. To understand why this program contains a bug, we must ask: *how does the value of `in_channels` relate to `x`'s shape?* PyTorch's documentation [PGM19] states that in the simplest case, the input to a convolution has the shape $(N, \texttt{in\_channels}, H, W)$. Indeed, in line 7, `x` is annotated with `TensorType([Dyn, Dyn, Dyn, Dyn]`, a typical gradual typing annotation indicating that `x` is a rank-4 tensor. The annotation does not state what the dimensions are, but it is still consistent with the shape stated in the documentation. Notice however that the `x`'s second dimension should match the value of `in_channels`. However, we have two values for `in_channels` that do not match. This mismatch will cause the program to crash if it ever receives any input, but not before. Our key questions can help us discover the bug statically across all inputs.

**Listing 5.2:** Gradually typed convolution

```
19 class ConvExample(torch.nn.Module):
```

```
20      def __init__(self):
21          super(BasicBlock, self).__init__()
22          self.conv1 = torch.nn.Conv2d(in_channels=2, ...)
23          self.conv2 = torch.nn.Conv2d(in_channels=4, ...)
24
25      def forward(self, x: TensorType([Dyn, Dyn, Dyn, Dyn])):
26          self.conv1(x)
27          return self.conv2(x)
```

By determining if we can replace all the `Dyn` dimensions in this program with numbers (which is the answer to Q(1) from our key questions), we can discover that it is impossible to assign a number to the second dimension of `x` and thus detect the error before running the program. More generally, the absence of a static typing sometimes can reveal that a program cannot run successfully on any input.

*How can we benefit from the migration space to answer Q(1) and thus detect that this program cannot be statically typed?* We can consider the migration space for this program. The space contains programs where `x` is a rank-4 tensor. A tool that can reason about the migration space can then take an extra constraint on the second dimension of `x`. The constraint should say that the second dimension must be a number. This constraint will narrow down the migration space to an empty set. The reason is that there is no such well-typed program. Therefore, we can conclude that the program cannot be statically typed because the second dimension cannot be assigned a number.

**Migration under arithmetic constraints** Let us fix the bug. One way to fix the bug is by removing `self.conv1` from the program. We get the program in Listing 5.3.

**Listing 5.3:** Gradually typed convolution

```
28 class ConvExample(torch.nn.Module):
29     def __init__(self):
30         super(BasicBlock, self).__init__()
31         self.conv2 = torch.nn.Conv2d(in_channels=4, ..)
32     def forward(self, x: TensorType([Dyn, Dyn, Dyn, Dyn])):
33         return self.conv2(x)
```

The program is correct and there can be various, correct ways to annotate it. The current annotation for the variable $x$ is that it is a tensor with four dimensions, but each dimension is denoted by `Dyn`, so the values of the dimensions are unknown. Suppose we want to specify constraints on those dimensions and determine if there are valid migrations that satisfy those constraints. This would be useful, not just for the user, but for compilers, since they can use those constraints to optimize for resources.

We can require some of the dimensions of `x` to be static and then provide arithmetic constraints on each of them. In this example, let us require all dimensions to be static. A tool can accept four constraints indicating this requirement. Then it can accept constraints that specify ranges on those dimensions. For example, the first dimension could be between 5 and 20. The second dimension can only have one possible value, which is 4. So it is enough to have a constraint requiring that dimension to be a number. The third dimension could also be between 5 and 20, while the fourth dimension could be between 2 and 10.

By giving these constraints as input to a tool, we are constraining the space to only the subspace that satisfies the constraints. A tool may find that this subspace indeed contains programs and outputs one of them. As a result, we may get the program in Listing 5.4. As shown, `x` has now been statically annotated with `TensorType([19, 4, 19, 9])`.

**Listing 5.4:** Statically typed convolution

```
34 class ConvExample(torch.nn.Module):
35     def __init__(self):
36         super(BasicBlock, self).__init__()
37         self.conv2 = torch.nn.Conv2d(in_channels=4, ..)
38     def forward(self, x: TensorType([19, 4, 19, 9])):
39         return self.conv2(x)
```

### 5.2.3 An example of branch elimination

**Listing 5.5:** Branch elimination

```
40 class ConvControlFlow(torch.nn.Module):
41    def __init__(self):
42        super().__init__()
43        self.conv = torch.nn.Conv2d(
44            in_channels=512, out_channels=512, kernel_size=3)
45
46    def forward(self, x: TensorType([Dyn, Dyn, Dyn, Dyn])):
47        if  self.conv(x).dim() == 4:
48            return torch.relu(x)
49        else:
50            return torch.nn.Dropout(x)
```

The program in Listing 5.5 is correct, but it contains control-flow in the form of a branch. We want to eliminate this branch. We refer to eliminating branches from a program by *branch elimination*. Eliminating branches enables programs to run on back-ends where branches are undesirable. `HFTracer` runs a program on a single input and computes the result of the branch and eliminates it accordingly. While the result of a branch could be fixed for

all program inputs, the result may also vary. Thus, running a program on just one input to eliminate a branch yields unsatisfactory branch elimination. We enable better branch elimination by finding all inputs for which a branch evaluates to a given result by reasoning about the program itself. We provide a mechanism to denote the set of inputs for which a branch evaluates to the given result. Notice that we reason about the static information given. Thus, if a variable has type `Dyn`, we optimistically assume that the program is well-typed and that the value for that variable will have the appropriate type at runtime.

The program in Listing 5.5 contains a condition that depends on shape information. This is a common situation, where ad-hoc shape-checks are inserted in a program to reason about its shapes. Line 8 has function that takes a variable `x` and applies a convolution to it, with `self.conv(x)`, and a condition that checks if the rank of `self.conv(x)` is 4. Since `x` is annotated as a rank-4 tensor on line 7, and convolution preserves the rank, `self.conv(x)` must also be rank-4. So the condition must always be true under the information given by `x`'s type annotation. We should be able to prove that the condition in line 8 always returns true without receiving any input for the program, by inspecting all the valid types that the program could possibly have. The migration space is useful for this analysis because it captures all possible, valid type annotations for a program.

Thus, under the convolution typing rules, if `self.conv(x).dim() == 4` is to evaluate to true, then `x` is also rank-4, which is consistent with `x`'s current annotation. In contrast, if `self.conv(x).dim() == 4` were to be false, i.e `self.conv(x).dim() != 4` is true, then this means that $x$ is not rank-4. However, the migration space of a program can never include inconsistent ranks for the same variable. Therefore, it is impossible to have `self.conv(x).dim() != 4`, while also having that `x` is rank-4.

A tool that reasons about the migration space as well as arbitrary predicates can make this conclusion. In this example, we can make a definitive conclusion about the result of this condition and we can re-write our program accordingly, as shown in Listing 5.6. This is a high-level intuition to a key idea, which we will expand on and formalize in Section 5.5. We will detail how we reason about the migration space in the presence of branches, and explain why our approach works.

**Listing 5.6:** Branch elimination

```
51 class ConvControlFlow(torch.nn.Module):
52     def __init__(self):
53         super().__init__()
54         self.conv = torch.nn.Conv2d(
55             in_channels=512, out_channels=512, kernel_size=3)
56 
57     def forward(self, x: TensorType([Dyn, Dyn, Dyn, Dyn])):
58         return torch.relu(x)
```

## 5.3 The Gradual Tensor Calculus

In this section, we describe our design choices then give the technical details of our type system. We finally prove gradual typing criteria for our system.

### 5.3.1 Design Choices

Our design choices are guided by enabling four key requirements: (1) modularity and backwards compatibility, (2) automatic tool support, (3) expressiveness, and (4) minimality

of our language. We have made these four choices in the context of tool support for PyTorch, but they can be extended to other frameworks. Here, we outline those design choices.

**Modularity and backwards compatibility** First, we require our system to support modularity and backwards compatibility for programs. A gradually typed system suits our needs because it supports partial type annotations. One of the implications of this support is that gradually typed programs can compile with any amount of type annotations. In a gradually typed system, a missing type is represented by the `Dyn` type.

The `Dyn` type can sometimes be assigned to a variable that has been used in different parts of the program with different, possibly inconsistent types. This type is useful when the underlying static type system is not flexible enough to fully type that program. For example, we may have a program that takes a batch of images with a dynamic batch size, as well as dynamic sizes, but with a fixed number of channels. In this case, a possible type would be $\texttt{TensorType}(\texttt{Dyn}, 3, \texttt{Dyn}, \texttt{Dyn})$, which indicates a batch of images, where the batch size is dynamic and the sizes are dynamic but the number of channels, which is 3, is fixed. Another example is that a variable could be assigned a rank-2 tensor at one point in the program, then a rank-3 tensor at a different point. A suitable type for that variable could simply be `Dyn`. In both examples, if we did not have the `Dyn` type, we would need more complex annotations. The `Dyn` type allows the gradual type checker to admit programs statically, and determine how to handle variables with `Dyn` types at runtime. The flexibility of gradual types stems from the consistency relation, which is symmetric and reflexive but not transitive. This relation allows a gradual type checker to statically admit programs in the absence of type information.

**Automatic tool support** Second, we require automatic tool support. We design a simple type system for a core language to enable us to define and solve problems for automatic tool support in a tractable way. Tool support is tractable because we followed a syntactic interpretation of gradual types. We base our approach on capturing the migration space by extending the constraint-based approach that is outlined in Chapter 3 to solve our three key questions.

**Expressiveness** Third, we require our system to be expressive enough to capture non-trivial programs. Our type system is more expressive than PyTorch's existing type-system, which does not reason about dimensions. Our language consists of a set of declarations followed by an expression. This structure is a convenient representation for the PyTorch neural network models we encountered, which mainly consisted of a function which takes a set of parameters. In the function body are tensor operations applied on those parameters. This calculus structure is inspired by the calculus from Rink [Rin18]. Rink highlighted that many DSLs can be mapped to their language. Besides adapting the structure of that calculus, we choose three core operations that present different challenges for automatic tool support, and then extend our support to 50 PyTorch operations.

**Minimality** Fourth, we require our language to be minimal so we can focus on our core problems. First, we do not introduce branches to our core grammar since, in practice, all tools on which we ran our experiments either do not accept programs with branches or aim to eliminate branches. As Reed et al. [RDH21] noted, many non-trivial tensor programs do not contain branches or statements.

Second, we do not consider runtime checks in gradual types. Those checks are typically a bottleneck for the performance of gradually typed programs [TFG16, GM18]. There has

been extensive research to alleviate performance issues by weakening these checks. As shown by Greenman and Fellisen [GF18], the notion of soundness in gradual types is not an all-or-nothing concept. Greenman and Felleisen [GF18] discuss three notions of soundness at different levels of strength and how they relate to performance: higher-order embedding of Tobin-Hochstadt and Felleisen [TF08], first-order embedding, as seen in Reticulated Python [VSS17] and erasure embedding, as seen in TypeScript [BAT14].

Similar to Rink [Rin18] and Reed et al. [RDH21], we observe that a language free from convoluted constructs represents a large subset of programs that are written in the machine learning area. As such, runtime errors are not as interesting when compared to those that arise in languages with constructs such as branches, functions, and function calls. Furthermore, runtime checks impose a computation cost on already costly tensor computations. Different compilers deal with runtime semantics in various ways. As a result, we choose to leave runtime aspects to future work.

### 5.3.2 Design Details

Figure 5.1 contains our gradual core calculus. We begin with a static calculus and gradualize it according to Cimini and Siek [CS16]. Our gradual calculus contains the dynamic type denoted by `Dyn`. A dimension can be `Dyn`, and a tensor can also be `Dyn`. A tensor is denoted by the constructor $\texttt{TensorType}(\sigma_1, ..., \sigma_n)$ where $\sigma_1, ..., \sigma_n$ are dimensions. However, if we denote a dimension by $U$ or $D$, it means the dimension is a number and cannot be `Dyn`. Our language has three expressions. $\texttt{add}(e_1, e_2)$ adds two tensors $e_1$ and $e_2$. $\texttt{reshape}(e, \tau)$ takes an expression $e$ and a shape $\tau$ and reshapes $e$ to a new tensor of shape $\tau$ if possible. Reshaping can be thought of as a re-arrangement of a tensor's elements. That requires the initial tensor to have the same number of elements as the reshaped tensor. $\tau$ can have

$$
\begin{array}{rrcl}
(Program) & p & ::= & \texttt{decl}^*\ \texttt{return}\ e \\
(Decl) & \texttt{decl} & ::= & x : \tau \\
(Terms) & e & ::= & x \mid \texttt{add}(e_1, e_2) \mid \texttt{reshape}(e, \tau) \mid \\
 & & & \texttt{Conv2D}(c_{in}, c_{out}, \kappa_{kernel}, e) \\
(IntegerTuple) & \kappa & ::= & (c^*) \\
(Const) & c & ::= & \langle \texttt{Nat} \rangle \\
(Tensor\ Types) & t, \tau & ::= & \texttt{Dyn} \mid \texttt{TensorType}(list(d)) \\
(Static\ Tensor\ Types) & S, T & ::= & \texttt{TensorType}(list(D)) \\
 & \sigma, d & ::= & \texttt{Dyn} \mid \langle \texttt{Nat} \rangle \\
 & U, D & ::= & \langle \texttt{Nat} \rangle \\
(Env) & \Gamma & ::= & \emptyset \mid \Gamma, x : \tau
\end{array}
$$

Notation:
$\delta$ is a sequence of dimensions with at most one occurrence of `Dyn`.

**Figure 5.1:** Gradual Tensor Core language

a maximum of one `Dyn` dimension. We will denote such a type by $\delta$ in our type checker. Finally we have $\texttt{Conv2D}(c_{in}, c_{out}, \kappa_{kernel}, e)$ which applies a convolution to $e$, given a number representing the input channel $c_{in}$, a number representing the output channel $c_{out}$, and a pair of numbers representing the kernel $\kappa_{kernel}$. The full version of convolution in PyTorch has more parameters. We have accounted for those parameters in our implementation, but because they create no new problems for us, our quest for minimality led us to leaving them out.

Figure 5.2 contains gradual typing relations that are used in our gradual typechecking. Those relations allow the typechecker to reason about the `Dyn` type. Matching, denoted by $\rhd$, and consistency, denoted by $\sim$, are standard in gradual typing and are lifted from equality

Consistency

$$\tau \sim \tau \ (c\text{-}refl\text{-}t) \quad d \sim d \ (c\text{-}refl\text{-}d)$$

$$d \sim \texttt{Dyn} \ (d\text{-}refl\text{-}dyn) \quad \tau \sim \texttt{Dyn} \ (t\text{-}refl\text{-}dyn)$$

$$\frac{\forall i \leq n : d_i \sim d'_i}{\texttt{TensorType}(d_1, ..., d_n) \sim \texttt{TensorType}(d'_1, ..., d'_n)} \ (c\text{-}tensor)$$

Type Precision

$$\tau \sqsubseteq \tau \ (refl\text{-}t) \quad d \sqsubseteq d \ (c\text{-}refl\text{-}d)$$

$$\texttt{Dyn} \sqsubseteq d \ (refl\text{-}dyn\text{-}1) \quad \texttt{Dyn} \sqsubseteq \tau \ (refl\text{-}dyn\text{-}2)$$

$$\frac{\forall i \leq n : d_i \sqsubseteq d'_i}{\texttt{TensorType}(d_1, ..., d_n) \sqsubseteq \texttt{TensorType}(d'_1, ..., d'_n)} \ (p\text{-}tensor)$$

Program and term Precision

$$\frac{\forall i \in \{1, ..., n\} \ \texttt{decl}'_i \sqsubseteq \texttt{decl}_i \quad e' \sqsubseteq e}{\texttt{decl}'_1, ..., \texttt{decl}'_n \ \texttt{return} \ e' \sqsubseteq \texttt{decl}_1, ..., \texttt{decl}_n \ \texttt{return} \ e} \ (p\text{-}prog) \quad \frac{\tau' \sqsubseteq \tau}{x : \tau' \sqsubseteq x : \tau} \ (p\text{-}decl)$$

$$e \sqsubseteq e \ (p\text{-}refl)$$

Matching

$$\texttt{TensorType}(\tau_1, ..., \tau_n) \rhd^n \texttt{TensorType}(\tau_1, ..., \tau_n)$$
$$\texttt{Dyn} \rhd^n \texttt{TensorType}(l) \text{ where } l = [\texttt{Dyn}, ..., \texttt{Dyn}] \text{ and } |l| = n$$

**Figure 5.2:** Auxiliary functions

in the static counter part of the system. Matching and consistency are both weaker than equality because they account for absent type information. Thus, if some type information is missing but there are no conflicting type information, matching and consistency would hold. Matching is a relation that pattern-matches two types. It is useful for arrow types in traditional type systems. Specifically, an arrow type $t_1 \to t_2$ matches itself. Type `Dyn` matches `Dyn` $\to$ `Dyn`. The ability to expand `Dyn` to become a function type `Dyn` $\to$ `Dyn` is valid in gradual types because it allows the system to optimistically consider the type `Dyn` to be `Dyn` $\to$ `Dyn`. We have adapted this definition to our system. First, we annotated matching with a number $n$ to denote the number of dimensions involved. So we have that $\texttt{TensorType}(\tau_1, ..., \tau_n) \rhd^n \texttt{TensorType}(\tau_1, ..., \tau_n)$ because any type matches itself. Similar to how traditionally, `Dyn` $\rhd$ `Dyn` $\to$ `Dyn`, we have that $\texttt{Dyn} \rhd^n \texttt{TensorType}(\texttt{Dyn}, ..., \texttt{Dyn})$, where $\texttt{Dyn}, ..., \texttt{Dyn}$ are exactly $n$ dimensions. Throughout this chapter, we will only use matching with $i = 4$ so we may use matching as $\rhd$ instead of $\rhd^4$. Consistency is a symmetric, reflexive, and non-transitive relation that checks that two types are equal, up to the known parts of the types. For example, the type `Dyn` contains no information, so it is consistent with any type, while the dimensions 3 and 4 are inconsistent because they are unequal. Figure 5.2 contains the formal definitions for matching and consistency.

$$\frac{}{\emptyset \vdash \bot} \; \textit{s-empty}$$

$$\frac{\texttt{decl}^* \vdash \Gamma \quad x \notin dom(\Gamma)}{\texttt{decl}^* \; id : \tau \vdash \Gamma : id \mapsto \tau} \; \textit{(s-var)}$$

**Figure 5.3:** Static context formation

Typing rules:

$$\frac{\texttt{decl}^* \vdash \Gamma \quad \Gamma \vdash e : \tau}{\vdash \texttt{decl}^* \ \texttt{return} \ e \ \texttt{ok}} \ \textit{(ok-prog)} \qquad \frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau} \ \textit{(t-var)}$$

$$\frac{\Gamma \vdash e : \texttt{TensorType}(D_1, ..., D_n) \quad \prod_1^n D_i = \prod_1^m U_i}{\Gamma \vdash \texttt{reshape}(e, \texttt{TensorType}(U_1, ..., U_m)) : \texttt{TensorType}(U_1, ..., U_m)} \ \textit{(t-reshape-s)}$$

$$\frac{\begin{array}{c}\Gamma \vdash e : \texttt{TensorType}(\sigma_1, ..., \sigma_m) \\ \prod_1^m \sigma_i \ mod \ \prod_1^n d_i = 0 \vee \prod_1^n d_i \ mod \ \prod_1^m \sigma_i = 0 \ \forall d_i, \sigma_i \neq \texttt{Dyn} \\ \textit{and } \texttt{Dyn} \textit{ occurs exactly once in } d_1, ...., d_m, \sigma_1, ..., \sigma_n \\ \textit{or} \\ \texttt{Dyn} \textit{ occurs more than once in } d_1, ...., d_m,\end{array}}{\Gamma \vdash \texttt{reshape}(e, \texttt{TensorType}(d_1, ..., d_n)) : \texttt{TensorType}(d_1, ..., d_n)} \ \textit{(t-reshape-g)}$$

$$\frac{\begin{array}{c}\Gamma \vdash e : \tau \ \textit{where} \\ \tau = \texttt{TensorType}(\sigma_1 \ ... \ \sigma_n) \\ \textit{and } \texttt{Dyn} \textit{ occurs more than once with at least one occurrence in} \\ \delta \ \textit{and } \sigma_1, ...., \sigma_m \\ \textit{or } \tau = \texttt{Dyn}\end{array}}{\Gamma \vdash \texttt{reshape}(e, \delta) : \delta} \ \textit{(t-reshape)}$$

$$\frac{\Gamma \vdash e : t \quad t \rhd^4 \texttt{TensorType}(\sigma_1, \sigma_2, \sigma_3, \sigma_4) \quad \tau = \texttt{calc-conv}(t, c_{out}, \kappa_{kernel}) \quad c_{in} \sim \sigma_2}{\Gamma \vdash \texttt{Conv2D}(c_{in}, c_{out}, \kappa_{kernel}, e) : \tau} \ \textit{(t-conv)}$$

$$\frac{\Gamma \vdash e_1 : t_1 \quad \Gamma \vdash e_1 : t_2 \quad (\tau_1, \tau_2) = \texttt{apply-broadcasting}(t_1, t_2) \quad \tau_1 \sim \tau_2}{\Gamma \vdash \texttt{add}(e_1, e_2) : \tau_1 \sqcap^* \tau_2} \ \textit{(t-add)}$$

**Figure 5.4:** Typing Rules

Figure 5.3 contains the static context formation rule, and figure 5.4 contains our typing rules. We need shorthands for some of our rules, which are contained in figure 5.5. Let us go over each type rule in detail. *ok-prog* and *t-var* are standard.

**Greatest lower bound:**

$$\tau \sqcap \tau' | \tau \nsim \tau' = \texttt{undefined}$$
$$\tau \sqcap \tau = \tau$$
$$\texttt{Dyn} \sqcap \tau = \tau$$
$$\tau \sqcap \texttt{Dyn} = \tau$$
$$\texttt{TensorType}(d_1, ..., d_n) \sqcap \texttt{TensorType}(d'_1, ..., d'_n) = \texttt{TensorType}(d_1 \sqcap d'_1, ..., d_n \sqcap d'_n)$$
$$d_1 \sqcap d_1 = d_1$$
$$d_1 \sqcap \texttt{Dyn} = d_1$$
$$\texttt{Dyn} \sqcap d_2 = d_2$$
$$d_1 \sqcap d_2 | d_1 \nsim d_2 = \texttt{undefined}$$

**Greatest lower bound *:**

$$\tau \sqcap^* \tau' | \tau \nsim \tau' = \texttt{undefined}$$
$$\texttt{Dyn} \sqcap^* \tau = \texttt{Dyn}$$
$$\tau \sqcap^* \texttt{Dyn} = \texttt{Dyn}$$
$$\texttt{TensorType}(d_1, ..., d_n) \sqcap^* \texttt{TensorType}(d'_1, ..., d'_n) = \texttt{TensorType}(d_1, ..., d_n) \sqcap \texttt{TensorType}(d'_1, ..., d'_n)$$

**apply-broadcasting:**

*apply-broadcasting*$(\tau_1, \tau_2)$ *is defined in the following way:*
If $\tau_1 = \texttt{Dyn} \vee \tau_2 = \texttt{Dyn}$, then return $\tau_1, \tau_2$.
Otherwise:

- Let $\tau_1$ and $\tau_2$ be equal in length by padding the shorter type with 1's from index 0.
- Replace occurrences of 1 in $\tau_1$ with the type at the same index in $\tau_2$.
- Replace occurrences of 1 in $\tau_2$ with the type at the same index in $\tau_1$.

**calc-conv:**

Let $t \rhd \texttt{TensorType}(\sigma_0, \sigma_1, \sigma_2, \sigma_3)$. Then
$\texttt{calc} - \texttt{conv}(t, c_{out}, \kappa_{kernel}) = \texttt{TensorType}(t'_0, t'_1, t'_2, t'_3)$ where:

$$t'_0 = \sigma_0$$
$$t'_1 = c_{out}$$
$$t'_2 = \begin{cases} \sigma_2 - 1 \times (\kappa_{kernel}[0] - 1) & \text{if } \sigma_2 \in \mathbb{N} \\ \texttt{Dyn} & \text{otherwise} \end{cases}$$
$$t'_3 = \begin{cases} \sigma_3 - 1 \times (\kappa_{kernel}[1] - 1) & \text{if } \sigma_3 \in \mathbb{N} \\ \texttt{Dyn} & \text{otherwise} \end{cases}$$



**Figure 5.5:** Broadcasting, convolution, and greatest lower bound

If two tensors $x$, $y$ are "broadcastable", let $broadcast(x, y)$ denote the resulting tensor shapes. The resulting tensor shapes are calculated as follows:

1. If the number of dimensions of $x$ and $y$ are not equal, prepend 1 to the dimensions of the tensor with fewer dimensions to make them equal length.
2. Then, for each dimension size, the resulting dimension size is the max of the shapes of $x$ and $y$ along that dimension.

**Figure 5.6:** Broadcasting runtime semantics

*t-reshape-s* is the static type rule for reshape. For reshape to succeed, the product of the dimensions of the input tensor shape must equal the product of dimensions of the desired shape. *t-reshape-g* assumes we have one missing dimension. Here, we can still determine if reshaping is possible using the modulo operation instead of multiplication. In this approach, we admit a program if we cannot prove it is ill-typed statically. *t-reshape* admits the expression if too many dimensions are missing.

To maintain minimality, *t-conv* deals with only the rank-4 case of convolution. *t-conv* expects a rank-4 tensor, so it uses matching ($\rhd^4$) to check the rank. Next, $c_{in}$ should be equal to the second dimension of the input, so the rule uses a consistency ($\sim$) check. Since the output of a convolution should also be rank-4, then apply `calc-conv` which, given a rank-4 input and the convolution parameters, computes the dimensions of the output shape. If a dimension is `Dyn`, then the corresponding output dimension will also be `Dyn`.

Finally, *t-add* adds two dimensions. Unlike scalar addition, the types of the operands do not have to be consistent. The reason is that broadcasting may take place. Broadcasting is a mechanism that considers two tensors and matches their dimensions. Two tensors are broadcastable if the following rules hold:

1. Each tensor has at least one dimension

2. When iterating over the dimension sizes, starting at the trailing dimension, the dimension sizes must either be equal, one of them is 1, or one of them does not exist

The runtime semantics of broadcasting are given by the PyTorch documentation. We include them in Figure 5.6

That tensors involved in broadcasting do not actually get modified to represent the modified shapes. This implies that the input shapes are not always consistent. Instead, the broadcasted result is only reflected in the output of the operation. Therefore, we have defined `apply-broadcasting` to simulate broadcasting on the inputs and consider what the types for these inputs would be, if broadcasting was to actually modify the inputs. In a static type system, the types of the modified inputs should be equal for addition to succeed. In gradual types, the types of the modified inputs should be consistent because equality lifts to consistency. We accomplish these requirements in our type rule. In particular, `apply-broadcasting` takes care of broadcasting the dimensions. Suppose that we are adding a tensor of shape $\texttt{TensorType}(\texttt{Dyn}, 2, \texttt{Dyn})$ to a tensor of size $\texttt{TensorType}(1, 2, 2)$. Then the output must be $\texttt{TensorType}(\texttt{Dyn}, 2, 2)$. The reason is that the first `Dyn` could be any number as per the broadcasting rules. So we cannot assume its value. The last dimension; however, must be 2 according to the rules. We have that:

$$\texttt{apply-broadcasting}(\texttt{TensorType}(\texttt{Dyn}, 2, \texttt{Dyn}), \texttt{TensorType}(1, 2, 2)) = \\ (\texttt{TensorType}(\texttt{Dyn}, 2, \texttt{Dyn}), \texttt{TensorType}(\texttt{Dyn}, 2, 2))$$

After simulating broadcasting, we may proceed as if we are dealing with regular addition. In other words, we check that the modified dimensions are consistent and get the greatest

lower bound. The greatest lower bound is a partial function which carries the dimensions of the addition operation. So we have that:

$$(\texttt{TensorType}(\texttt{Dyn}, 2, \texttt{Dyn}) \sqcap \texttt{TensorType}(\texttt{Dyn}, 2, 2)) = \texttt{TensorType}(\texttt{Dyn}, 2, 2)$$

We will cover one last special case for addition. Simply applying the greatest lower bound to the modified input types of addition is not general enough to cover the following case. Suppose we are adding a tensor of shape `Dyn` to a tensor of shape $\texttt{TensorType}(1, 2)$, then we must output `Dyn` because the output type could be a range of possibilities. In this case, `apply-broadcasting` does not modify the types because the tensor of shape `Dyn` could range over many possibilities. We then apply our modified version of the greatest lower bound denoted by $\sqcap^*$, which behaves exactly like $\sqcap$ except when one of the inputs is `Dyn`, where it returns `Dyn` to get that:

$$\texttt{TensorType}(1, 2) \sqcap^* \texttt{Dyn} = \texttt{Dyn}$$

### 5.3.3 Example of how type checking works

Let consider an untyped version of Listing 5.2, which we will call Listing 5.7.

**Listing 5.7:** An example of incompatible dimension uses in `Conv2D`

```
59 class ConvExample(torch.nn.Module):
60     def __init__(self):
61         super(BasicBlock, self).__init__()
62         self.conv1 = torch.nn.Conv2d(in_channels=2, out_channels=2,
       kernel_size=2, ...)
```

```
63          self.conv2 = torch.nn.Conv2d(in_channels=4, out_channels=2,
        kernel_size=2, ...)
64
65      def forward(self, x: Dyn):
66          self.conv1(x)
67          return self.conv2(x)
```

Consider `self.conv1(x)` and `self.conv2(x)`. Looking at our core calculus again, we have that a convolution is of the form $\texttt{Conv2D}(c_{in}, c_{out}, \kappa_{kernel}, e)$. So `self.conv1(x)` can be expressed as $\texttt{Conv2D}(2, 2, (2, 2), x)$ in our calculus, while `self.conv2(x)` can be expressed as $\texttt{Conv2D}(4, 2, (2, 2), x)$ in our calculus (modulo the extra parameters such as `padding` and `stride`, which we replaced by "...", and which do not affect the types of each of those expressions).

The derivation tree for `self.conv1(x)` would be of the following form:

$$\frac{\begin{array}{c}\Gamma \vdash x : \texttt{Dyn}\ \ \texttt{Dyn} \rhd^4 \texttt{TensorType}(\texttt{Dyn}, \texttt{Dyn}, \texttt{Dyn}, \texttt{Dyn}) \\ \tau = \texttt{calc-conv}(\texttt{Dyn}, 2, (2, 2))\ \ 2 \sim \texttt{Dyn}\end{array}}{\Gamma \vdash \texttt{Conv2D}(2, 2, (2, 2), x) : \texttt{TensorType}(\texttt{Dyn}, 2, \texttt{Dyn}, \texttt{Dyn})}\ \textit{(t-conv)}$$

We have a similar derivation tree for `self.conv2(x)` which has the form:

$$\frac{\begin{array}{c}\Gamma \vdash x : \texttt{Dyn}\ \ \texttt{Dyn} \rhd^4 \texttt{TensorType}(\texttt{Dyn}, \texttt{Dyn}, \texttt{Dyn}, \texttt{Dyn}) \\ \tau = \texttt{calc-conv}(\texttt{Dyn}, 4, (2, 2))\ \ 4 \sim \texttt{Dyn}\end{array}}{\Gamma \vdash \texttt{Conv2D}(4, 2, (2, 2), x) : \texttt{TensorType}(\texttt{Dyn}, 2, \texttt{Dyn}, \texttt{Dyn})}\ \textit{(t-conv)}$$

In the derivation tree for `self.conv1`, we have that $c_{in} \sim \texttt{Dyn}$, where $c_{in} = 2$. This corresponds to `in_channels=2`. In the derivation tree for `self.conv2`, we have that $c_{in} = 4$,

which corresponds to `in_channels=4`. So observe that in the full derivation tree of the program, we would have a consistency check of the form $\sigma \sim 4$ and another consistency check of the form $\sigma \sim 2$, where $\sigma$ refers to the second dimension of the shape of $x$.

As we know from the discussion in Section 5.2, even though we can migrate the `Dyn` type of $x$ to a rank-4 tensor with $\texttt{TensorType}(\texttt{Dyn}, \texttt{Dyn}, \texttt{Dyn}, \texttt{Dyn})$, the second dimension of $x$ must remain to be `Dyn` because of the fact that $4 \sim \sigma$ and $2 \sim \sigma$ both occur in the derivation tree. We will expand on this example in Section 5.4 to answer Q(1).

### 5.3.4 The Gradual Criteria

We prove that our type system satisfies standard criteria from Siek et al. [SVC15a] that are relevant to this project. Those are static criteria that concern the migration space. Since we are not considering runtime aspects, we will leave the theorems about those aspects to future work. First, we prove the static gradual guarantee, which describes the structure of the migration space. The conservative extension theorem shows that our gradual calculus subsumes its static counter-part. We use standard notations for our theorems. The full definitions and proofs can be found in the Appendix .3.

We adapt the criteria to our calculus and re-write the Monotonicity w.r.t precision theorem in the following way:

**Theorem 5.3.1** (Monotonicity w.r.t precision)**.** $\forall p, p' : \mathit{if}\ \vdash p : \mathit{ok}\ \wedge\ p' \sqsubseteq p\ \mathit{then} \vdash p' : \mathit{ok}.$

We have that gradual typing is a conservative extension of a static calculus. We denote a well-typed program in the statically typed tensor calculus by $\vdash_S p : \texttt{ok}$. The statically typed tensor calculus is defined in Appendix .1.

**Theorem 5.3.2** (Conservative Extension)**.** *For all static p, we have:*
$\vdash_S p : \mathit{ok}$ *iff* $\vdash p : \mathit{ok}$

## 5.4 The Migration Problem as a constraint satisfiability problem

A migration is a more static, well-typed version of a program. We can define that $P'$ is a migration of $P$ (written $P \leq P'$) iff $(P \sqsubseteq P' \wedge \vdash P' : \texttt{ok})$. Notice that our programs are always closed. However, for an open program, one can adapt a more general definition of the form $P$ (written $P \leq_\Gamma P'$) iff $(P \sqsubseteq P' \wedge \Gamma \vdash P' : \texttt{ok})$ and $\vdash P' : \texttt{ok}$ can be written as $\emptyset \vdash P' : \texttt{ok}$.

Intuitively, this means that we are annotating a program with more static types such that the program continues to type check. All migrations of a given program form the *Migration Space*, denoted by $Mig(P)$.

Our goal is to capture the migration space via constraints, in the same way that we have in Chapter 3. Every solution to our constraints for a program must map to a corresponding migration for the same program. In other words, one satisfying assignment to the constraints results in one migration.

In previous work such as [SMS18], constrained based approaches have been used. In this chapter, we also follow a constrained based approach. Specifically, We follow the same approach outlined in Chapter 3, which involves defining constraints whose solutions are order-isomorphic with the migration space. However, due to the arithmetic nature of our constraints, our solution procedure is different and we use an SMT solver to find a satisfying assignment, which would equate to finding a migration. Later in this chapter, we will show how to use this framework to answer our three key questions.

### 5.4.1 Our source grammar, target grammar, and encoding the Tensor type and the Dynamic type

We have two grammars of constraints. A high-level, source grammar, shown in figure 5.7, that is used for generating the constraints. That grammar gets translated to low-level constraints, which are drawn from our target grammar from Figure 5.9. Having two grammars is not necessary, but it makes the constraint generation process more tractable and simplifies the presentation. We can view the source grammar as syntactic sugar for the target grammar. Our target constraints can be understood by an SMT solver.

Let us expand on both grammars. Our source constraint grammar which can be found in figure 5.7 contains constraints that capture all well-typed programs that are more precise than the current program. We will explain what those constraints mean in detail, but let us present an overview. Our grammar consists of precision constraints of the for $\tau \sqsubseteq x$. $x$ here indicates a type variable for the variable $x$ from the program. Thus, $x$ in the constraint $\tau \sqsubseteq x$ captures all types that are more precise than $\tau$. We also have matching constraints of the form $[\![e]\!] \rhd \texttt{TensorType}(\delta_1, \delta_2, \delta_3, \delta_4)$, consistency constraints of the form $D \sim \delta, \langle e \rangle \sim \langle e \rangle$ and greatest lower bound constraints of the form $\langle e \rangle \sqcap^* \langle e \rangle$. Those are gradual typing constraints that we use to faithfully model our gradual typing rules. Our constraint grammar also contains short-hands such as `can-reshape`$([\![e]\!], \delta)$ and `apply-broadcasting`$([\![e]\!], [\![e]\!])$. Those short-hands represent the typing rules as well. `can-reshape` expands to further constraints which evaluate to true if $[\![e]\!]$ can be reshaped to $\delta$. Similarly, when expanded, `apply-broadcasting`$([\![e]\!], [\![e]\!])$ captures all possible ways to broadcast two types. Because we prioritize tractability of the migration space, we set the upper bound of tensor ranks to 4, via a constraint of the form $[\![e]\!] \leq 4$. We make this decision because in practice, all benchmarks we considered did not have tensors with ranks exceeding this number.

Since our constraints involve gradual types, let us describe how we encoded types so that they can be understood by an SMT solver. Because we fixed an upper bound for tensor ranks, we chose to encode tensor types as uninterpreted functions, which means that we have a constructor for each of our ranks, of the form `TensorType1`, `TensorType2`, `TensorType3`, and `TensorType4`. Each of the functions take a list of dimensions. Moving on to the dimensions, we have that dimensions are either `Dyn` or natural numbers. We can easily represent natural numbers in an SMT solver but we must also represent `Dyn`. One way to encode a `Dyn` dimension $d$ is as a pair $(d_1, d_2)$. If $d_1 = 0$, then $d = \texttt{Dyn}$. Otherwise, $d$ is a number, and its value is in $d_2$. Let us formalize the constraint generation process next.

### 5.4.2 Constraint Generation

From $p$, we generate constraints $Gen(p)$ as follows. Let $p$ have the form $\texttt{decl}^*\ \texttt{return}\ e$. Assume that $p$ has been $\alpha$-converted so that all declared variables are distinct from each other. Let $X$ be the set of declaration-variables $x$ occurring in $e$, and let $Y$ be a set of variables disjoint from $X$ consisting of a variable $[\![e']\!]$ for every occurrence of the subterm $e'$ in $e$. Let $Z$ be a set of variables disjoint from $X$ and $Y$ consisting of a variable $\langle e_1 \rangle, \langle e_2 \rangle$ for every occurrence of the subterm $\texttt{add}(e_1, e_2)$ in $e$. Finally, let $V$ be a set of variables disjoint from $X$, $Y$, and $Z$ consisting of dimension variables $\zeta$. The notations $[\![e]\!]$ and $\langle e \rangle$ are ambiguous because there may be more than one occurrence of some subterm $e'$ in $e$ or some subterm $e''$ in $e$. However, it will always be clear from context which occurrence is meant. For every occurrence of $\zeta$, it is implicit that we have a constraint $\zeta \geq 0$ to prevent the solver from assigning dimensions outside of $\mathbb{N}$. We omit writing this explicitly at every step of our constraint generation and resolution to avoid redundancy. With that in mind, we generate the constraints in figure 5.8.

$$
\begin{aligned}
(\mathit{Constraints})\quad \psi \;::=\;& \psi \wedge \psi \mid \psi \vee \psi \mid \texttt{True} \mid \texttt{False} \\
&\mid [\![x]\!] = x \mid [\![e]\!] = \tau \mid \tau \sqsubseteq x \\
&\mid |[\![e]\!]| \leq 4 \mid D \sim \delta \mid \langle e \rangle \sim \langle e \rangle \\
&\mid [\![e]\!] \rhd \texttt{TensorType}(\zeta_1, \zeta_2, \zeta_3, \zeta_4) \\
&\mid [\![e]\!] = \langle e \rangle \sqcap^* \langle e \rangle \mid \texttt{can-reshape}([\![e]\!], \delta) \\
&\mid [\![e]\!] = \texttt{calc-conv}([\![e]\!], c_{out}, \kappa_{kernel}) \\
&\mid \langle e \rangle, \langle e \rangle = \texttt{apply-broadcasting}([\![e]\!], [\![e]\!])
\end{aligned}
$$

**Figure 5.7:** Source constraint grammar

$$\frac{}{\vdash x : \tau : \tau \sqsubseteq x \wedge |x| \leq 4}\ (t\text{-}decl) \qquad \frac{}{\Gamma \vdash x : x = [\![x]\!]}\ (t\text{-}var)$$

$$\frac{\Gamma \vdash e : \psi}{\Gamma \vdash \texttt{reshape}(e, \delta) : \psi \wedge\ [\![\texttt{reshape}(e, \delta)]\!] = \delta \wedge \texttt{can-reshape}([\![e]\!], \delta) \wedge |[\![e]\!]| \leq 4}\ (t\text{-}reshape)$$

$$\frac{\Gamma \vdash e : \psi}{\begin{array}{c}\Gamma \vdash \texttt{Conv2D}(c_{in}, c_{out}, \kappa_{kernel}, e) : \psi\ \ \wedge [\![e]\!] \rhd \texttt{TensorType}(\zeta_1, \zeta_2, \zeta_3, \zeta_4) \wedge c_{in} \sim \zeta_2\ \wedge \\ [\![\texttt{Conv2D}(c_{in}, c_{out}, \kappa_{kernel}, e)]\!] = \texttt{calc-conv}([\![e]\!], c_{out}, \kappa_{kernel})\end{array}}\ (t\text{-}conv)$$

$$\frac{\Gamma \vdash e_1 : \psi_1\ \ \Gamma \vdash e_2 : \psi_2}{\begin{array}{c}\Gamma \vdash \texttt{add}(e_1, e_2) : \psi_1 \wedge \psi_2 \wedge [\![\texttt{add}(e_1, e_2)]\!] = \langle e_1 \rangle \sqcap^* \langle e_2 \rangle\ \wedge \\ (\langle e_1 \rangle, \langle e_2 \rangle) = \texttt{apply-broadcasting}([\![e_1]\!], [\![e_2]\!]) \wedge \langle e_1 \rangle \sim \langle e_2 \rangle\ \wedge \\ |[\![e_1]\!]| \leq 4 \wedge |[\![e_2]\!]| \leq 4 \wedge |[\![\texttt{add}(e_1, e_2)]\!]| \leq 4\end{array}}\ (t\text{-}add)$$

**Figure 5.8:** Constraint generation

Let us go over the rules in figure 5.8.

*t-decl* uses the precision relation $\sqsubseteq$ to insure that a migration will have a more precise type, while *t-var* propagates the type information from declarations to the program.

*t-reshape* considers all possibilities of reshaping any tensor $e$ with rank, at most 4, via the constraint $[\![e]\!] \leq 4$. This restriction constraint captures all rank possibilities for $[\![e]\!]$ in addition to $[\![e]\!]$ being `Dyn`. For each possibility, the number of occurrences of `Dyn` in $\delta$ and $[\![e]\!]$ varies. This impacts the arithmetic constraints that make reshaping possible, as we can see from the typing rules. As such, *can-reshape* simulates all such possibilities and generates the appropriate constraints.

*t-conv* contains matching and consistency constraints, to model matching and consistency in convolution's typing rule. We have a constraint `calc-conv`, which generates the appropriate arithmetic constraints for the output of the convolution, based on the convolution typing rule, again accounting for the possibility of the input $e$ having a gradual type.

*t-add* contains greatest lower bound constraints and consistency constraints, similar to the add typing rule. We constrain the inputs $e_1$and $e_2$, as well as the expression itself, $add(e_1, e_2)$ to all be either `Dyn` or tensor of at most rank-4, via a $\leq$ constraint. We use the function `apply-broadcasting`, which simulates broadcasting on the shapes, on dummy variables $\langle e_1 \rangle$ and $\langle e_2 \rangle$ (notice that the real shapes of $e_1$ and $e_2$ are represented by $[\![e_1]\!]$ and $[\![e_2]\!]$). At this point, we can check $\langle e_1 \rangle$ and $\langle e_2 \rangle$ for consistency and obtain the greatest lower bound, as per our typing rules.

### 5.4.3 Constraint Resolution

Let us describe our target grammar. We define $\texttt{IntConst} = \mathbb{N}$, and we use $n$ to range over `IntConst`. We use $v$ as a meta variable. It ranges over variables which range over $\texttt{TensorType}(list(\zeta)) \cup \{Dyn\}$ and we use $\zeta$ as a meta variable that ranges over variables that range over $\texttt{IntConst} \cup \{Dyn\}$. This grammar is useful for our constraint resolution process. In particular, the first step of solving our constraints is to translate them to low-level constraints, drawn from our target grammar, before feeding them to an SMT solver. The process of translating high-level constraints to our target constraints is detailed in Appendix .4.

$$
\begin{array}{rrcl}
(\mathit{Constraints}) & \psi & ::= & \psi \wedge \psi \mid \psi \vee \psi \mid \neg\psi \mid \texttt{True} \mid \texttt{False} \\
& & \mid & v = \texttt{TensorType}(\zeta, ..., \zeta) \mid v = \texttt{Dyn} \mid v = v \\
& & \mid & \zeta = n \mid \zeta = \texttt{Dyn} \mid \zeta = \zeta \mid \zeta = \zeta * n + n \\
& & \mid & (\zeta_1 * ... * \zeta_m) \; mod \; (\zeta'_1 * ... * \zeta'_n) = 0
\end{array}
$$

**Figure 5.9:** Target constraint grammar

We define a solution $\varphi$ as follows.

| For each: | we have: |
|---|---|
| $\psi \wedge \psi'$ | $\varphi \vDash \psi \wedge \psi'$ |
| $\psi \vee \psi'$ | $\varphi \vDash \psi \vee \psi'$ |
| $\neg\psi$ | $\varphi \vDash \neg\psi$ |
| $\texttt{True}$ | $\varphi \vDash \texttt{True}$ |
| $\texttt{False}$ | $\varphi \vDash \texttt{False}$ |
| $v = \texttt{TensorType}(\zeta_1, ...\zeta_n)$ | $\varphi(v) = \texttt{TensorType}(\varphi(\zeta_1), ...\varphi(\zeta_n))$ |
| $v = \texttt{Dyn}$ | $\varphi(v) = \texttt{Dyn}$ |
| $v = v'$ | $\varphi(v) = \varphi(v')$ |
| $\zeta = n$ | $\varphi(\zeta) = n$ |
| $\zeta = \texttt{Dyn}$ | $\varphi(\zeta) = \texttt{Dyn}$ |
| $\zeta = \zeta'$ | $\varphi(\zeta) = \varphi(\zeta')$ |
| $\zeta = \zeta * n + n'$ | $\varphi(\zeta) = \varphi(\zeta') * n + n'$ |
| $(\zeta_1 * ... * \zeta_m) \; mod \; (\zeta'_1 * ... * \zeta'_n) = 0$ | $(\varphi(\zeta_1) * ... * \varphi(\zeta_m)) \; mod \; (\varphi(\zeta'_1) * ... * \varphi(\zeta'_n)) = 0$ |

Given a program $P$, automatic code annotation works as follows. Let $Solve$ be function that takes a set of constraints and invokes an SMT solver, returning a mapping $res : x \times \tau$ from type variables to shapes.

1. Let $C = Gen(p)$
2. $Solve(c) = res$
3. For every variable $x \in dom(res)$, we have that $res(x) = t$ where $t$ is a shape.

Given $P$, let $res$ be the output of an SMT solver. Define $reannotate$ as follows:

1. $reannotate(p, res) = reannotate(decl^*, res)p$
2. $reannotate(decl_1, ..., decl_n, res) = reannotate(decl_1, res), ..., reannotate(decl_n, res)$
3. $reannotate(id : \tau, res) = id : \tau'$ if $id : \tau \in res$
4. $reannotate(id : \tau, res) = undefined$ if $id \not\in dom(res)$

**Figure 5.10:** An algorithm for automatic code annotation

Let $Gen(P)$ be the constraint generation function and $Sol(C)$ be the set of solutions to constraints $C$. Then we can state the order-isomorphism theorem as follows:

**Theorem 5.4.1** (Order-Isomorphism)**.** $\forall P, \Gamma : (Mig(P), \sqsubseteq)$ *and* $(Sol(Gen(P)), \leq)$ *are order-isomorphic.*

The order-isomorphism theorem states that we have captured the migration-space with our constraints such that, for a given program, the solution space and the migration-space are order-isomorphic. For the proof, see Appendix .5.

Our algorithm for automatic code annotation is shown in Figure 5.10.

### 5.4.4 A Migration Example

Let us now revisit Listing 5.7 and show how to migrate it by generating our constraints and passing them to an SMT solver.

Let us recall that this listing had two expressions that map to $\texttt{Conv2D}(2, 2, (2, 2), x)$ and $\texttt{Conv2D}(4, 2, (2, 2), x)$.

The first step is to generate high-level constraints: a conjunction of the following constraints.

$$\texttt{Dyn} \sqsubseteq v_1 \tag{1}$$

$$v_1 \leq 4 \tag{2}$$

$$v_1 \rhd \texttt{TensorType}(\zeta_3, \zeta_4, \zeta_5, \zeta_6) \tag{3}$$

$$2 \sim \zeta_4 \tag{4}$$

$$v_2 = \texttt{calc-conv}(v_1, 2, (2, 2), (2, 2), (2, 2), (2, 2)) \tag{5}$$

$$0 \leq \zeta_3 \wedge 0 \leq \zeta_4 \wedge 0 \leq \zeta_5 \wedge 0 \leq \zeta_6 \tag{6}$$

$$v_1 \rhd \texttt{TensorType}(\zeta_9, \zeta_{10}, \zeta_{11}, \zeta_{12}) \tag{7}$$

$$4 \sim \zeta_{10} \tag{8}$$

$$v_8 = \texttt{calc-conv}(v_1, 2, (2, 2), (2, 2), (2, 2), (2, 2)) \tag{9}$$

$$0 \leq \zeta_9 \wedge 0 \leq \zeta_{10} \wedge 0 \leq \zeta_{11} \wedge 0 \leq \zeta_{12} \tag{10}$$

Let us go over what each equation is for. Constraint (1) denotes that the type annotation for the variable $x$ must be as precise or more precise than `Dyn`. Constraint (2) denotes that the type annotation for $x$ could either be `Dyn` or a tensor with at most four dimensions. We

use the $\leq$ notation to denote this. Notice that the type variable for $x$ is $v_1$. Constraints (3), (4), (5) and (6) are for $\texttt{Conv2D}(2, 2, (2, 2), x)$, while constraints (7), (8), (9) and (10) are for $\texttt{Conv2D}(4, 2, (2, 2), x)$. More specifically, constraints (3) and (7) determine the input shape of a convolution while constraints (5) and (9) determine the output shape of a convolution. Notice that constraints (6) and (10) are dimension constraints which indicate that dimensions are natural numbers.

The main differences between the constraints for our core calculus and the ones in our implementation is that `calc-conv` takes some additional parameters in our implementation because we have implemented the full version of convolution.

The constraints above are high-level constraints which are yet to be expanded. For example, $\rhd$ and $\leq$ constraints get transformed to equality constraints. We will skip writing out the resulting constraints for simplicity. After expanding these constraints and running them through an SMT solver, we get satisfying assignments. The fact that we got a satisfying assignment lets us know that the migration space is non-empty, which means that the program is well-typed. Let us go through some of relevant assignments:

$$\varphi(v_1) = \texttt{Dyn}$$
$$\varphi(v_2) = \texttt{TensorType}(\texttt{Dyn}, 2, \texttt{Dyn}, \texttt{Dyn})$$
$$\varphi(v_8) = \texttt{TensorType}(\texttt{Dyn}, 2, \texttt{Dyn}, \texttt{Dyn}))$$

Here, $v_1$ is the type of $x$, $v_2$ is the type of the first convolution and $v_8$ is the type of the second convolution. We can see that these assignments are a valid typing to the program because the outputs of both convolutions should be 4-dimensional tensors with the second dimension being 2, which stands for the output channel. And since the input $x$ has been assigned `Dyn` by our SMT solver, we cannot determine the last two dimensions of a convolution output.

While this is a reasonable output, it may not be helpful to the programmer. Furthermore, this program would not accept any concrete output. We know this from our constraints. From constraints (3) and (7), we have that $\sigma_4 = \sigma_{10}$. Then from (4), (8), which are $2 \sim \sigma_4$ and $4 \sim \sigma_{10}$, we can see that the only satisfying solution is `Dyn`. This means that the program cannot be statically typed. Next, we will see how to prove this formally.

### 5.4.5 Capturing a subset of the migration space to solve Q(1) and Q(2)

Let us discuss how to extend our approach to solve Q(1) and Q(2). In the example above, the migration space is non-empty and we may want to know if we can statically type the program. We have established that we cannot. As a first step, we may want to take our core constraints above, which we will call $C$, and restrict the input to a rank-4 tensor. So we can consider the constraint $C \wedge x = \texttt{TensorType}(\zeta_1', \zeta_2', \zeta_3', \zeta_4')$ where $\zeta_1', ..., \zeta_4'$ are fresh variables. We can begin to impose restrictions on $\zeta_1', ..., \zeta_4'$ to make them concrete variables. For example, if we restrict the last dimension to be a number, we can add the constraint $\zeta_4' \neq \texttt{Dyn}$. After running our constraints through the solver, we get the following assignments:

$$\varphi(v_1) = \texttt{TensorType}(\texttt{Dyn}, \texttt{Dyn}, \texttt{Dyn}, 28470)$$
$$\varphi(v_2) = \texttt{TensorType}(\texttt{Dyn}, 2, \texttt{Dyn}, 14236)$$
$$\varphi(v_8) = \texttt{TensorType}(\texttt{Dyn}, 2, \texttt{Dyn}, 14236)$$

To prove that no concrete assignment to the second dimension of $x$ is possible, we simply add $\zeta_2' \neq \texttt{Dyn}$ to our original constraints and the constraints will be unsatisfiable, so we conclude that the second dimension of $x$ can only be `Dyn`.

We can also answer Q(2) by feeding the solver additional arithmetic constraints about dimensions. In our example, if we want the first dimension of $x$ to be between 3 and 10, we can add the constraint $\zeta'_1 <= 3 \wedge \zeta'_1 >= 10$ to $C \wedge x = \texttt{TensorType}(\zeta'_1, \zeta'_2, \zeta'_3, \zeta'_4)$ and rerun our solver.

### 5.4.6 Decidability and Complexity of Migration

Our migration solution is based on a satisfiability problem. We have two questions to consider. First, *Is our migration problem decidable?* and second *If so, what is the time complexity?*.

The migration problem is satisfiable if the underlying constraints are drawn from a decidable theory. Those underlying constraints are the ones given by the grammar in Section 5.4.3. Let us for a moment ignore constraints of the form $(\zeta_1 * ... * \zeta_m)\ mod\ (\zeta'_1 * ... * \zeta'_n) = 0$. We observe that all the other constraints are drawn from a well-known decidable theory. Specifically, the other constraints are drawn from quantifier-free Presburger arithmetic extended with uninterpreted functions and equality. The satisfiability problem for this theory is NP-complete [SB05]. Once we add constraints of the form $(\zeta_1 * ... * \zeta_m)\ mod\ (\zeta'_1 * ... * \zeta'_n) = 0$, the decidability-status of the satisfiability problem is unknown, to the best of our knowledge. Fortunately, only three operations need this additional constraint: `Reshape`, `View`, or `Flatten`. All the other 47 operations that our implementation supports need only constraints in the NP-complete subset. We have summarized the operations encountered in practice in figure 5.13. Our implementation translates all of the constraints to Z3 format, and while our benchmarks do need constraints outside the NP-complete subset, our experiments terminated.

The complexity of migration depends on the size of the constraint we generate. The bottleneck in our constraint generation process is the $\leq$ constraint. Let us see how to expand it.

$$From : |[\![e]\!]| \leq 4$$

$$To : [\![e]\!] = \texttt{Dyn} \vee [\![e]\!] = \texttt{TensorType}(\zeta_1) \vee ... \vee [\![e]\!] = \texttt{TensorType}(\zeta_1, \zeta_2, \zeta_3, \zeta_4)$$

where $\zeta_1, ..., \zeta_4$ are fresh variables

This yields a complexity of $4^n$ in the number of $\leq$ constraints. So assuming that any additional constraints are drawn from the NP-complete subset, the problem will still be decidable. Note that if we are working with a fixed rank, then these constraints will be generated in polynomial time in the size of the program. We will see how even solving the problem for a fixed rank has practical benefits in the next section.

## 5.5 Extending our approach to solve Q(3): Branch Elimination

In example 5.5 from Section 5.2, we considered a conditional that depended on the rank of the input. In example 5.8, we consider a slightly more complicated example, where the conditional depends on the value of one of the dimensions in the input shape.

**Listing 5.8:** An example of graph-break elimination

```
68          class ReshapeControlFlow(torch.nn.Module):
69              def __init__(self):
70                  super().__init__()
```

```
71
72              def forward(self, x: Dyn):
73                  if x.reshape(100).size()[0] < 100:
74                      return torch.dropout(x, p=0.5, train=False)
75                  else:
76                      return torch.relu(x)
```

The example uses the `reshape` function. Recall that reshape is an operation that takes a tensor and re-arranges its elements according to the desired shape. In this case, we are reshaping `x` to have the shape `TensorType`([100]). For reshaping to succeed, the initial tensor must contain the same number of elements as the reshaped tensor. Let us establish that since `x` is typed as `Dyn`, the program will type check. In the expression `x.reshape(100).size()`, `size()` will return the shape of `x.reshape(100)`, which is [100]. We are then getting the first dimension of the shape in the expression `x.reshape(100).size()[0]`, which is 100. Thus, by inspecting the conditional `if x.reshape(100).size()[0] < 100`, we can see that the conditional should always evaluate to false. With this information, the tracer can remove branches from the program and continue the tracing process to obtain a full trace.

In `TorchDynamo`, Listing 5.8 gets broken into two different programs. One for when the condition evaluates to true, and another for when the condition evaluates to false. But intuitively, the condition could be eliminated to result in the program in Listing 5.9. Let us see an example of how extend our constraint based solution to eliminate the extra branch. We then formalize our approach.

**Listing 5.9:** An example of graph-break elimination

```
77          class ReshapeControlFlow(torch.nn.Module):
78              def __init__(self):
```

```
79                 super().__init__()
80
81             def forward(self, x: Dyn):
82                 return torch.relu(x)
```

First, we can generate the core constraints for our program up to the point of encountering a branch. `x.reshape(100)` is the same as `reshape(x, TensorType(100))` from our core calculus.

$$\texttt{Dyn} \sqsubseteq v_1 \tag{1}$$

$$v_1 \leq 4 \tag{2}$$

$$v_2 = \texttt{TensorType}(100) \tag{3}$$

$$\texttt{can-reshape}(v_1, \texttt{TensorType}(100)) \tag{4}$$

$$v_2 = v_3 \tag{5}$$

$$(v_3 = \texttt{Dyn} \wedge \zeta_4 = \texttt{Dyn}) \vee \tag{6}$$

$$((\zeta_4 = \texttt{GetItem}(v_3, 1, 0) \vee \zeta_4 = \texttt{GetItem}(v_3, 2, 0) \vee$$

$$\zeta_4 = \texttt{GetItem}(v_3, 3, 0) \vee \zeta_4 = \texttt{GetItem}(v_3, 4, 0)])) \wedge$$

$$(0 \leq \zeta_4)$$

We extend our constraint grammar with constructs that enable us to represent `size()` and indexing into shapes. This includes constraints such as `GetItem`. We discuss this point in Section 5.6.

Let us discuss the constraints. Constraints (1) and (2) are for $x$. Notice that $v_1$ is the type variable for $x$. Constraints (3) and (4) are for `reshape(x, TensorType(100))`. Next, when encountering the `size` function in a program, we simply propagate the shape at hand with an equality constraint, which is seen in equation (5). If we are indexing into a shape, we consider all the possibilities for the sizes of that shape and generate constraints accordingly. In particular, we have that ($v_3 = \texttt{Dyn} \wedge \zeta_4 = \texttt{Dyn}$) because a shape could be dynamic, which means that if we index into it, we get a `Dyn` dimension. But since we restricted our rank to 4, we can consider the possibilities of the index being 1, 2, 3 or 4, which is what the remaining constraints do. `GetItem` expands to simpler constraints described by the same grammar above, but intuitively, $\texttt{GetItem}(v, c, i)$ generates constraints where $v$ is the shape we are indexing into, $c$ is the assumed tensor rank, and $i$ is the index of the element we want to get. If we have a disjunction of `GetItem` constraints for every possible rank, then we can capture all possibilities for what $\zeta_4$ can be.

After capturing the migration space up to the point of a conditional, we must also generate a constraint for the conditional and its negation. More elaborately, we can generate the constraint $\zeta_4 < 100$, where $\zeta_4$ holds the first dimension of the reshaping result. Namely, `x.reshape(100).size()[0]`. So denote the set of constraints we generated by $C$. Then we consider $C \wedge \zeta_4 < 100$ and $C \wedge \neg(\zeta_4 < 100)$. We evaluate both sets of constraints. One set must be satisfiable while the other must be unsatisfiable for us to conclude a result. If we are unable to do so, this means that the input set is still too general such that for some inputs, the branch may evaluate to true and for other inputs, the branch may evaluate to false. In such case, we can ask the user to capture a stricter subset of the input by further constraining it. We can then re-evaluate our constraints again to see if we are able to make a decision.

Figure 5.11 captures a simplified format of the predicates that appear in branches within PyTorch programs. Predicates thus involve constants and type variables. Our constraints

$$
\begin{array}{rrcl}
(Predicate) & \mathcal{P} & ::= & \mathcal{T} > \mathcal{T} \mid \mathcal{T} \leq \mathcal{T} \mid \mathcal{T} = \mathcal{T} \\
(TypeVariable) & \mathcal{T} & ::= & \tau \mid < \texttt{Var} >
\end{array}
$$

**Figure 5.11:** Constraint grammar for predicates

reason about a given program to represent its migration space up to the point of encountering a conditional. We represent the conditional as a constraint which acts as a restriction on the migration space to capture part of it. While branch elimination eliminates branches correctly according to the type information that is statically available, it is not semantics preserving. More specifically, because we optimistically assume that if a variable has a type `Dyn`, that it will have the appropriate value at runtime, if we could remove a branch that would trigger a runtime type error. However, we believe that this issue could be tackled using gradual runtime semantics, which we leave to future work. Next, let us elaborate on how we utilize the migration space for branch elimination. Consider the sequence $e\ \mathcal{P}$ where $e$ is an expression followed by a predicate $\mathcal{P}$. Then generate constraints for $e$ in the way we described in Section 5.4 and denote them by $C$. Then, consider the predicate $\mathcal{P}$. This predicate is an additional constraint. Let us denote it by $C'$. Let $s_1 = C \wedge C'$ and $s_2 = C \wedge \neg C'$. Solve the constraints by following the steps in 5.4.3. If $s_1$ is satisfiable and $s_2$ is unsatisfiable, then the result of the predicate is false. If $s_1$ is unsatisfiable and $s_2$ is satisfiable, then the result of the predicate is true. Notice that we have two cases missing, which is when $s_1$ is satisfiable and $s_2$ is satisfiable. This entails that the condition can evaluate to true or false, depending on the input. Thus, the input has not been constrained enough by the current constraints on the migration space to make a definitive conclusion. If both $s_1$ and $s_2$ evaluate to false, then

the constraints imposed by predicate $\mathcal{P}$ on the migration space caused it to be empty, which means that the program is ill-typed under that predicate. Notice that the above grammar expands on the source constraint grammar we had before. However, it is straightforward to translate the terms of this grammar to Z3. So, the translation was a straightforward extension.

## 5.6 Implementation

### 5.6.1 The setting: Three PyTorch tracers

PyTorch has three tool-kits that rely on symbolic tracers. Let us go over each one. First, `torch.fx` [RDH21] is a common PyTorch tool-kit and has a symbolic tracer. Symbolic tracing is a process of extracting a more specialized program representation from a program, for the purpose of analysis, optimization, serialization, etc. `torch.fx` does not accept programs containing branches. `HFtracer` [WDS20] eliminates branches by symbolically executing on a single input. Finally, `TorchDynamo` [Ans22] handles dynamic shapes by dividing the program into fragments. This process is called a *graph-break*. Specifically, when encountering a condition that depends on shape information and where shape information is unknown, the program is broken into two parts. One fragment is for when the result of the condition is true, and another is for when the result of the condition is false. Graph-breaks result in multiple programs with no branches.

As a technical detail automatic code annotation for the purpose of program understanding and better documentation is meant to be performed on a source language; branch elimination is done in trace-time, on an intermediate representation. For the purpose of better readability, we presented all the examples in Section 5.2 in source code syntax. In some of our larger

benchmarks, the source code is different from the intermediate representation because more high-level constructs were used, such as statements. However, statements do not influence our theoretical results. Similarly, some of the examples below contain sequences of expressions. We did not include sequences in our theory because they did not introduce additional challenges to our problem. Finally, there are some constructs in PyTorch that propagate variable shapes, such as dim() and size(). There are also getters which index into shapes. Those constructs were used to write ad-hoc shape-checks. We dealt with them in our implementation by propagating shape information accordingly.

### 5.6.2 Implementation details

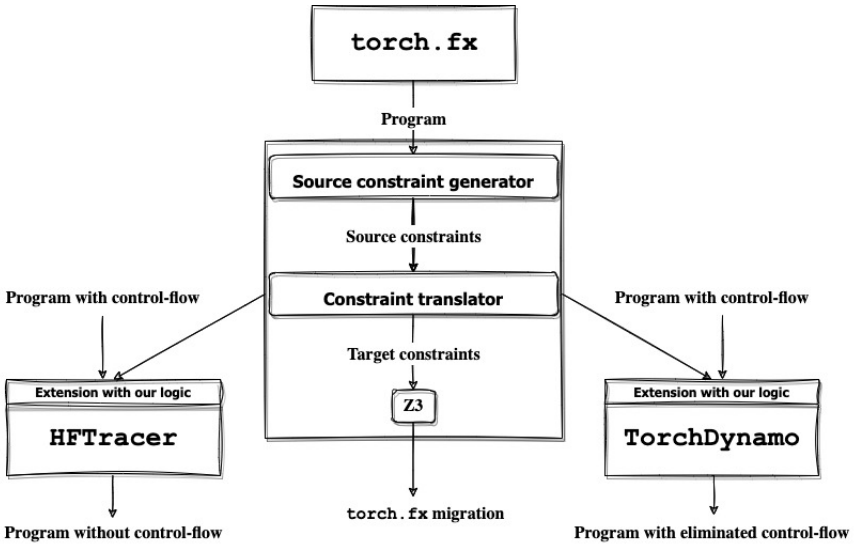


**Figure 5.12:** Our core tool and the three tracers

We have implemented approximately 6000 LOC across three different tracers. Figure 5.12 summarizes how our implementation works. First, we implement a core constraint

generator. This generator takes a program (in our benchmarks case, a program is generated via `torch.fx`), and generates core, source constraints for it. Next is the constraint translator which consists of two phases. In the first phase, it encodes the gradual types found in the constraints then translates the source constraints into target constraints. In the second phase, it translates the target constraints into Z3 constraints. This is a 1:1 translation. The constraints are then fed into Z3 to generate a migration (in our benchmarks, it is a `torch.fx` migration).

Next, we modify each of `TorchDynamo` and `HFtracer` to incorporate our reasoning and use it for branch elimination. We must incorporate our logic into the tracers because *branch elimination happens in trace-time*, unlike program migration which requires a whole program.

Our implementation faithfully follows our core logic, although we have made some practical simplifications. First, there were operations where some edge cases did not exist in any of our benchmarks, so we skipped implementing them. Second, for the `view` operation, we have skipped implementing dynamism and required the solver to provide concrete dimensions. This allowed us to carry out branch elimination without requiring an additional constraint that disables dynamism, although the same effect can be accomplished in this manner as well. Third, `conv2D` may accept rank-3 or rank-4 inputs, but we have limited our implementation to the rank-4 case, since this is the case that is relevant to most of our benchmarks.

We have chosen Z3 as a constraint solver because it is efficient, can represent our constraints and has a Python interface which enables ease of use. Our tool accepts custom user-constraints, which can be used to answer different questions about the migration space.

**Platform.** We ran our experiments on a Mac Book Pro with an 8-Core CPU, 14-Core GPU and 512GB DRAM.

## 5.7 Experimental Results

**Questions and claims.** We answer the following three questions

Q(1) Can our tool determine if the migration space is non-empty? If so, can it determine if the migration space contains a static migration and if so, can it find one? *Yes. Our tool is the first to affirmatively answer all three questions.*

Q(2) Given an arithmetic constraint on a dimension, can our tool determine if there a migration that satisfies it and if so, can it find one? *Yes. Our tool is the first to retrieve migrations that provably satisfy arbitrary arithmetic constraints.*

Q(3) Can our tool prove that branch elimination is valid for an infinite set of inputs, not just for a single input? If so, does it allow us to represent the set of inputs for which a branch evaluates to true or false? *Yes. We incorporate our logic into two different tools and eliminate branches in all benchmarks we considered for infinite classes of input, which we characterized via constraints. Neither tool was able to achieve this without our logic.*

### 5.7.1 Benchmark suite description

Figure 5.13 contains our benchmark names, lines of code and the number of `flatten` and `reshape` operations in each benchmark. Those operations are special because our analysis on them involves multiplication and modulo constraints.

**Where did we get our benchmarks?** ConvExample is the example in Listing 5.7, while BmmExample is the program that was inspired by the paper [PS21] and involves an incorrect use of batch matrix multiplication. ResNet50 and AlexNet are popular machine learning

models. The remaining benchmarks are transformer models [Wol00]. Those open-source models are popular in the machine learning area.

**Why did we use different benchmarks for different experiments?** The first four models do not contain branches, making them suitable for Q(1) and Q(2). The next 6 models are average-size models that are suitable for our `HFTracer` experiments. Those experiments required reasoning about whole programs and our tool was able to reason about them in under two minutes. The final five benchmarks are of a larger size. We do not support all the operations in those benchmarks. However, this did not pose a problem because in `TorchDynamo`, we were not required to reason about entire programs. Instead, we were required to reason about program fragments, which made our tool terminate in under three minutes. Those benchmarks would cause performance slowdowns if reasoned about as whole programs. We will discuss this later in detail.

### 5.7.2 Does a program have a static migration?

**How we can run our tool to answer Q(1)**

1. Generate the core constraints and check if they are satisfiable. If not, stop right away; The program is ill-typed.

2. Determine if the input variable can have a concrete rank by asking the solver for migrations of concrete ranks from one to four. If none exist, the input variable was used at different ranks throughout the program.

3. If the input variable can be assigned concrete ranks, pick one of them and ask the tool to statically annotate all dimensions.

| Benchmark | LOC | Flatten | Reshape |
|---|---|---|---|
| BmmExample | 4 | 0 | 0 |
| ConvExample | 6 | 0 | 0 |
| AlexNet | 24 | 1 | 0 |
| ResNet50 | 177 | 1 | 0 |
| XGLM | 104 | 0 | 14 |
| Electra | 525 | 0 | 0 |
| Roberta | 533 | 0 | 0 |
| MobileBert | 2103 | 0 | 0 |
| Bert | 528 | 0 | 0 |
| MegatronBert | 1018 | 0 | 0 |
| MarianMT | 1735 | 0 | 315 |
| Marian | 1733 | 0 | 315 |
| M2M100 | 1762 | 0 | 319 |
| BlenderBot | 2380 | 0 | 451 |

**Figure 5.13:** General benchmark information

| Benchmark | Static migration? | Time(s) |
|---|---|---|
| BmmExample | No | 0.03 |
| ConvExample | No | 0.05 |
| AlexNet | Yes | 2 |
| ResNet50 | Yes | 5 |

**Figure 5.14:** Q(1): Static migration

4. If the solver cannot statically annotate all dimensions, relax this requirement for each dimension to determine which one cannot be statically annotated.

We first traced our benchmarks using `torch.fx`, then ran the above steps on the output. The first step simply involves running our tool, while the second and third steps require the user to pass constraints to the tool and rerun it. Determining if a variable has a certain rank requires a single run with our tool. Determining if a dimension can be static requires a single run with our tool. The final step involves removing constraints. Each time we remove a constraint from a dimension, we can run our tool once to determine a result.

Figure 5.14 summarizes our results. The first column in the figure is the benchmark name. The second column asks if the benchmark has a static migration and the third column measures the time it took to answer this question and retrieve a static migration. Below is a summary of the results.

For ConvExample, the input can only be rank-4 and the second dimension can only be `Dyn`. BmmExample has a type error. Finally, ResNet50 and AlexNet can be fully typed and the inputs can only be rank-4 in both cases.

| Benchmark | Arithmetic constraints? | Time(s) |
|---|---|---|
| BmmExample | No | 0.03 |
| ConvExample | Yes | 0.08 |
| AlexNet | Yes | 2 |
| ResNet50 | Yes | 347 |

**Figure 5.15:** Q(2): Migration under arithmetic constraints

### 5.7.3 Can we retrieve migrations that satisfy arithmetic constraints?

**How we ran our tool to answer Q(2)**

1. Follow the steps for answering Q(1)
2. If any dimensions can be static, then we apply further arithmetic constraints on some of those dimensions and ask for a migration that satisfies them.

We ran the steps above in our extension of `torch.fx`. Step (2) requires one run with our tool. Our results are summarized in figure 5.15. The first column is the benchmark name. The second column asks if arithmetic constraints can be imposed on at least one of the dimensions and the third column measures the time it took to answer this question and retrieve a migration that satisfies an arithmetic constraint. Below is a summary of the results.

For ResNet50 and AlexNet, we added arithmetic constraints. For ConvExample, we fixed the example like we did in Section 5.2 then added arithmetic constraints. We obtained valid migrations that satisfy our constraints for all benchmarks, except for BmmExample which is ill-typed and thus has an empty migration space. Notice that our tool can reason about any arithmetic constraints that can be accepted by Z3.

### 5.7.4 Can we perform branch elimination on an infinite class of inputs?

**How to answer Q(3)** We ran our extension of `HFtracer`, starting with annotating the input with `Dyn` and then gradually increasing the precision of our constraints to provide the solver with more information to eliminate more branches. The number of times we run our tool here depends on how much information the user gives the tool about the input. If the tool receives static input dimensions, then this will be enough to eliminate all branches. But since we aim to relax this requirement, we could start with a `Dyn` shape then gradually impose constraints, first with rank information, then with dimension information.

We were able to eliminate all branches this way. We followed similar steps in our `TorchDynamo` extension but we faced some practical concerns because `TorchDynamo` currently does not carry parameter information between program fragments. We had to resolve this issue manually by passing additional constraints at every new program fragment.

Consider figures 5.16 and 5.17 which detail our `HFtracer` experiments on 6 workloads. Figure 5.16 contains the original number of branches in the program, the remaining branches after running our extension, without imposing any constraints on the input, and the number of remaining branches after running our extension, with constraints on the input. The last column of the figure is the time it takes to perform branch elimination with constraints.

`HFtracer` also eliminates all branches from the 6 workloads. However, it does this by running the program on an input. We can obtain a similar result by giving a constraint describing the *shape* of the input because we observed that for all benchmarks we considered, an actual input is not needed to eliminate all branches, and we can relax this requirement much further. Specifically, in some benchmarks, no constraints are needed at all to eliminate all branches. While for some of them, it is enough to specify rank information. For one of the benchmarks, we can specify a range of dimensions for which branches can be eliminated.

Figure 5.17 details our results. It contains the benchmarks and the constraints we gave our tool so that it could eliminate all branches from each benchmark.

Finally figure 5.18 represents branch elimination for `TorchDynamo`. Let us discuss what the existing tracer does. There are two modes of operation in `TorchDynamo` called static and dynamic. In the static mode, the tracer traces the program with one input which is provided by the user. Branch elimination is therefore valid for a single input. In Dynamic mode, the tracer also takes an input but it only records *rank* information and ignores the values of the dimensions. So if a branch depends on dimension information, a graph-break will occur.

We focused on benchmarks where branches depend on dimension information. In figure 5.18, we impose constraints on the dimensions and eliminate branches which decreases the number of times `TorchDynamo` breaks the program when tracing. The first column in the figure indicates the benchmark names. Next is the original number of branches with `TorchDynamo`. Then we have the remaining number of branches after incorporating our reasoning. Finally, we measure time in seconds. The input constraints are range and rank constraints; we omit the details.

### 5.7.5 Limitations

**Performance limitations.** From our four experiments, we observed that slowdowns can be due to factors including the kind of constraints involved and the number of constraints to solve. Our tool typically handles benchmarks that are under 1000 lines of code most efficiently. However, range constraints impose overhead. For example, ResNet50 and XGLM contain such constraints and they were the slowest in figures 5.15 and 5.16. The benchmarks in figure 5.18 were over 1000 lines and we have observed performance issues where we had to

| **Benchmark** | original | w/o constraints | with constraints | Time(s) |
|---|---|---|---|---|
| XGLM | 5 | 4 | 0 | 22 |
| Electra | 3 | 3 | 0 | 1 |
| Roberta | 3 | 0 | 0 | 3 |
| MobileBert | 3 | 3 | 0 | 1 |
| Bert | 3 | 0 | 0 | 3 |
| MegatronBert | 3 | 0 | 0 | 5 |

**Figure 5.16:** Q(3): `HFtracer` number of remaining branches

skip certain program fragments, which exceeded 5 minutes to terminate. On average, one can expect a typical benchmark to contain approximately 1000 lines of code.

**Implementation limitations.** There are two limitations to our `TorchDynamo` experiments. First, since PyTorch has various operations with many layers of abstractions and edge cases, not every edge case was implemented. Given that this only affected a few branches, we chose to skip those branches. This did not affect our experiments because `TorchDynamo` does not require all branches to be removed. Each branch removed will result in one less graph-break. `TorchDynamo` induces graph-breaks for reasons other than control flow. When graph-breaks happen, we have to re-write an input constraint for the resulting fragments because there is currently no clear mechanism in passing parameter information from one fragment to another. We manually passed input constraints to program fragments until eliminating at least 40% of branches and have stopped after that due to the large size of the benchmarks and program fragments. We leave parameter information preservation during graph-breaks to the `TorchDynamo` developers.

| **Benchmark** | our constraint |
|---|---|
| XGLM | $Tensor(x, y)\|x > 0, y < 2000$ |
| Electra | $Tensor(x, y)$ |
| Roberta | No constraint |
| MobileBert | $Tensor(x, y)$ |
| Bert | No constraint |
| MegatronBert | No constraint |

**Figure 5.17:** Q(3): `HFtracer` constraints on inputs for which branch elimination is valid

| **Benchmark** | original | with constraints | Time(s) |
|---|---|---|---|
| XGLM | 5 | 0 | 45 |
| MarianMT | 44 | 26 | 75 |
| M2M100 | 47 | 22 | 130 |
| Marian | 44 | 26 | 70 |
| BlenderBot | 35 | 19 | 40 |

**Figure 5.18:** Q(3): `TorchDynamo` number of remaining branches

## 5.8 Related work

### 5.8.1 Related work about shapes in tensor programs

We first discuss related work about shapes in tensor programs.

**Tensors Fitting Perfectly** Paszke and Brennan [PS21] show how to do shape checking based on assertions written by programmers. Their assertions can reason about tensor ranks and dimensions, with arithmetic constraints. Our work also supports such constraints. Their tool executes a program symbolically and looks for assertion violations. The more assertions programmers write, the more shape errors their tool can report. Their tool uses Z3 to solve constraints of a size that can be up to exponential in the size of the program. Our approach is similar in that it enables programmers to annotate a program with types and to type check the program and thereby catch shape errors. Another similarity is that we use Z3 to solve constraints of exponential size. Our approach differs by going further: we can automatically annotate a program with types and we can automatically remove unnecessary runtime shape checks. Additionally, we have proved that our type system has key correctness properties.

**Gradual Tensor Shape Checking** Hattori et al. [HKS22] focuses on shape checking and shape inference, while we focus on generalizing shape analysis for various tasks including program migration and branch elimination. Hattori et al. [HKS22] define a gradually typed system for tensor computations and, like us, they prove that it has key correctness properties. They use refinement types to represent tensor shapes, they enable programmers to write type annotations, and they do best-effort shape inference. Their refinements share some characteristics with the assertions used by Paszke and Brennan [PS21], as well as with our constraints. Their approach *adds* the traditional gradual runtime checks [ST06] in cases

where annotations and shape inference fall short. Our work differs by enabling automatic program optimizations through removing runtime checks, while we leave adding gradual runtime checks to future work. Conceptually, our approach and the one from Hattori et al. [HKS22] differ significantly. Firstly, as in Chapter 3, we follow a syntactic interpretation of gradual types and consider type migration as a syntactic definition. Hattori et al. [HKS22] follows a semantic interpretation of gradual types. It is unclear how migration would be defined in this context. Another difference is that we have demonstrated scalability: their benchmark programs are up to 258 lines of code, while our benchmark programs are up to 2,380 lines of code.

**An Empirical Study on TensorFlow Program Bugs** Zhang et al. [ZCC18] analyzed the root causes of bugs in TensorFlow programs by scanning StackOverFlow and GitHub. They identified four symptoms and seven root causes for such bugs. The most common symptoms are functional errors, crashes, and build failure, while common root causes are data processing errors, type confusion, and dimension mismatches. Our type system can help spot those root causes because key parts of such code will have type `Dyn`, even after migration.

**Static Analysis of Shape in TensorFlow Programs** Lagouvardos et al. [LDG20] use static analysis to detect shape errors in TensorFlow. Their approach statically detects 11 of the 14 shape-related TensorFlow bugs reported by Zhang et al. [ZCC18]. However, they prove no correctness properties. Our approach differs from Lagouvardos et al. [LDG20] by being able to automatically annotate a program with types and being able to automatically remove unnecessary runtime checks. Our work can reason about programs without requiring any type annotations and only taking into account the shape information from the operations used in the program, while Lagouvardos et al. [LDG20] requires a degree of type information. For

example, looking at the constraints for `Reshape`, we can see that their constraint generator may require a tensor rank, while our work does not rely on this assumption, because in PyTorch, such information may not be statically available. Lagouvardos et al. [LDG20] focus on TensorFlow, while we focus on PyTorch. The two languages have some semantic differences. We leave extending our work to other languages to future work. Finally, in contrast to their work, we have proved that our type system has key migratory properties, such as that our constraints represent the entire migration space for a program, allowing us to extract and reason about all existing shape information from the program according to the programmer's needs.

**ShapeFlow** Verma et al. [VS20] consider a dynamic analysis tool called ShapeFlow to detect shape errors. It focuses on TensorFlow. The advantage of this approach is that, like our approach, it does not require type annotations, but their analysis holds for only particular inputs, in contrast to our approach, which reasons about programs across all possible inputs. Unlike our work, the approach used in ShapeFlow has not been formalized, but there is empirical evidence to support that it detects shape errors in *most* cases. Because we reason about programs statically, our work is more suitable for compiler optimizations and program understanding. Our shape analysis approach can be used to automatically annotate programs. In contrast, ShapeFlow is more suitable if a programmer desires a light-weight form for error detection that works in most cases.

**Relay** Roesch et al. [RLW18] designed an intermediate representation called Relay. It is functional, similar to our calculus, but is statically-typed, unlike our type system, which is gradually typed. Its goals are similar to ours in that it aims to balance between various properties such as expressiveness, portability, and compilation. Unlike our system, as a static

type system, Relay requires type annotations for every function parameter. Similar to our approach, their work focuses on the static aspect of the problem and has left the runtime aspect to future work.

The work by Roesch et al. [RLK19] extends the work by Roesch et al. [RLW18]. It uses a static polymorphic type system for shapes. One of the main differences is that this system also has a type named `Any`. Even though this type may enable partial annotations, it is unclear whether this type has the same flexibility as our `Dyn` type, which is part of gradually typed systems. Gradually typed systems enable type consistency, while statically typed systems do not. Type consistency enables us to type check programs which may not be well-typed in a statically typed system. Since the underlying system in relay is a static type system, it is unclear whether the type `any` provides the same level of flexibility. Finally, we have not considered polymorphic types and we leave it to future work.

Pytea [JKS22] is a static analysis tool that detects shape errors. Their approach is different than ours in that it detects errors via symbolic execution. It considers all possible execution paths for a program to reason about shapes. The number of execution paths can be large. In contrast, our approach reasons about shapes which can be given in the form of type annotations or can be detected from the program.

### 5.8.2 Related work about migratory typing

There are various works about migratory typing. We will discuss the most two related ones.

**What is Decidable about Gradual Types?** In Chapter 3, we defined the migration space for a gradually typed program as the set of all well-typed, more-precise programs. We represented the migration space for a given program by generating constraints where

each solution represents a migration. The constraint-based approach enables us to give algorithms for solving migration problems for a $\lambda$-calculus. In this chapter, we have followed a syntactic interpretation of gradual types, which enabled us to adopt Chapter 3's definition for type migration and the migration space. Therefore, in our context of a tensor calculus and rather different types, we use their idea of a migration space and constraints to give an algorithm that automatically annotates a program with types and an algorithm that removes unnecessary runtime checks. In contrast to their approach, we use an SMT solver because it can deal with the arithmetic nature of tensor constraints.

**TypeWhich** Phipps-Costin et al. [PAG21] build a tool which extends on the work from Chapter 3, by providing several criteria for choosing migrations from the migration space. Their work is about simple types, while our work is about tensor shapes. While their work is specifically focused on reasoning about the migration space for program annotation, we reason about the migration space more generally, by using it for general tensor reasoning tasks including program annotation and branch elimination. Their gradual language contains traditional gradual runtime checks, which they may take into consideration when migrating programs, while we leave runtime aspects to future work.

## 5.9 Conclusion

We have presented a method that reasons about tensor shapes in a general way. Our method involves a gradual tensor calculus that satisfies key properties and supports decidable shape analysis for a large set of operations. We have shown that our algorithm is practical by showing that it works on 15 non-trivial benchmarks across three different tracers.

# CHAPTER 6

# Conclusion

## 6.1 Summary

Developers need automated type annotation tools to cope with the migration of untyped programs to partially or fully typed ones. My dissertation presents three different models of automatic migration for three different scenarios.

Specifically, for the first scenario present in Chapter 3, we provided a foundation for better tool support by settling decidability questions about migration with gradual types for the Gradually Typed Lambda Calculus (GTLC) of Siek and Taha [ST06]. We presented three algorithms and a hardness result for deciding key properties and we explained how they can be useful during an exploration. In particular, we showed how to decide whether the migration space is finite, whether it has a top element, and whether it is a singleton. We also showed that deciding whether it has a maximal element is NP-hard.

In Chapter 4, our goal was to support type migration for intersection types while simultaneously satisfying most of Siek et al.'s gradual typing criteria [SVC15a]. The result is a system which conservatively extends the GTLC with Rank-2 gradual intersection types and can type strictly more programs than the GTLC but lacks certain properties that typically accompany intersection types such idempotence, commutativity and associativity as a trade-off. When it comes to gradual typing criteria, the system also lacks type soundness. However, it satisfies

the remaining criteria. In our system, we showed how to decide whether the migration space is finite, whether it has a top element, and whether it is a singleton at the same time complexities that we did for the GTLC.

In Chapter 5, we considered machine learning frameworks such as PyTorch [PGC17] where tensors are the central data structure. We recognized that shape information is useful for preventing shape mismatches, providing better documentation and readability as well as improving program capture. For this scenario, we designed a tensor based language and an accompanying gradually type system, then we considered our constraint based approach from above to solve several problems about migrating programs in this setting. The problems we considered were (1) how to find a static migration, (2) how to find a migration that satisfies an arithmetic constraint, and (3) how to eliminate branches that depend on input shapes for an infinite class of inputs. We addressed all three problems and provided experimental results that show improvement over existing PyTorch tools.

## 6.2 Future Work

In this work, we present results concerning automatic type migration for three scenarios. Future work can pursue automatic type migration that supports subtyping [ST07, VKS14, GCT16, TF08], refinement types [LT17, TF08], monotonic references [SVC15b], and set theoretic types [CLP19, TF08].

In Chapter 4, we explore the Rank-2 system that supports three decision problems for automatic type migration but misses some of the gradual typing criteria of Siek et al. [SVC15a]. Thus, future work can also explore how to design systems that support automatic type migration while simultaneously satisfying the gradual typing criteria of Siek et al [SVC15a].

As shown in Chapter 3, the concept of a migration space can be adapted to different settings, but the specific questions about the migration space may vary. Thus, future work can consider more questions of the migration space, which can aid the developer in their exploration during automatic code annotation.

Finally, which criteria should we require of a gradually typed system? In our work, we aim to satisfy various criteria depending on the setting. In Chapter 3, our system satisfies migration criteria as well as all refinement criteria of Siek et al. Given that the GTLC is quite small, it is straightforward to satisfy all criteria. In Chapter 4, we face challenges when trying to satisfy the same criteria, which leads us to enforcing several language restrictions, in an effort to reach a trade-off between criteria that we can satisfy and the expressiveness of our language. In Chapter 5, we tweak our migration criteria to fit the specific scenario at hand, because we observe that different migration questions apply to different settings. We also decide to opt for erasure semantics and notice that this choice was acceptable for our results. Since efficiency is a crucial element in machine learning today, future work can explore the trade-offs between adding runtime checks to the program and sacrificing efficient versus type safety. More generally, future work can rethink migratory and gradual critrea that a language must satisfy for a particular setting.

## 6.3 Final Remarks

In summary, my dissertation validates that algorithmic type migration poses interesting and challenging questions. I demonstrate this point with three major case studies, ranging from a simple core language to a useful framework language for machine learning.

Indeed, as this last result shows, concerns about algorithmic type migration should be taken into account during the design of the gradual type system. My dissertation's results

show that it is factor of equal weight to many others and that doing so yields immediate benefits.

# APPENDIX A

# What is Decidable about Gradual Types?

## A.1 Proof of the Unique Type Theorem

We will prove Theorem 3.1, which we restate here:

$\forall E, \Gamma, T, T'$, if $\Gamma \vdash E : T$ and $\Gamma \vdash E : T'$, then $T = T'$.

*Proof.* We proceed by induction on the derivation of $\Gamma \vdash E : T$. We will do a case analysis based on the last rule that was used to derive $\Gamma \vdash E : T$.

Case: *T-Num*. The last rule that was used to derive $\Gamma \vdash E : T'$ must be *T-Num*, so $T = \texttt{int} = T'$.

Case: *T-True*. The last rule that was used to derive $\Gamma \vdash E : T'$ must be *T-True*, so $T = \texttt{bool} = T'$.

Case: *T-False*. The last rule that was used to derive $\Gamma \vdash E : T'$ must be *T-False*, so $T = \texttt{bool} = T'$.

Case: *T-Var*. The last rule that was used to derive $\Gamma \vdash E : T'$ must be *T-Var*, so $T = \Gamma(x) = T'$.

Case: *T-Abs*. The last step of the derivation of $\Gamma \vdash E : T$ must be as follows:

$$\frac{\Gamma, x : T_1 \vdash F : T_2}{\Gamma \vdash (\lambda x : T_1.F) : T_1 \to T_2} \textit{ (T-Abs)}$$

. The last rule that was used to derive $\Gamma \vdash E : T'$ must be *T-Abs*, as follows:

$$\frac{\Gamma, x : T_1 \vdash F : T_2'}{\Gamma \vdash (\lambda x : T_1.F) : T_1 \to T_2'} \textit{ (T-Abs)}$$

. For the subderivations $\Gamma, x : T_1 \vdash F : T_2$ and $\Gamma, x : T_1 \vdash F : T_2'$, we apply the Induction Hypothesis to get that $T_2 = T_2'$. We conclude $T_1 \to T_2 = T_1 \to T_2'$.

Case: *T-App*. The last step of the derivation of $\Gamma \vdash E : T$ must be as follows:

$$\frac{\begin{array}{cc}\Gamma \vdash E_1 : T_1 & \Gamma \vdash E_2 : T_2 \\ T_1 \rhd (T_{11} \to T_{12}) & T_2 \sim T_{11}\end{array}}{\Gamma \vdash E_1\ E_2 : T_{12}} \textit{ (T-App)}$$

The last rule that was used to derive $\Gamma \vdash E : T'$ must be *T-App*, as follows:

$$\frac{\begin{array}{cc}\Gamma \vdash E_1 : T_1' & \Gamma \vdash E_2 : T_2' \\ T_1' \rhd (T_{11}' \to T_{12}') & T_2' \sim T_{11}'\end{array}}{\Gamma \vdash E_1\ E_2 : T_{12}'} \textit{ (T-App)}$$

For the subderivations $\Gamma \vdash E_1 : T_1$ and $\Gamma \vdash E_1 : T_1'$, we apply the Induction Hypothesis to get that $T_1 = T_1'$. From $T_1 = T_1'$ and $T_1 \rhd (T_{11} \to T_{12})$ and $T_1' \rhd (T_{11}' \to T_{12}')$, we conclude that $(T_{11} \to T_{12}) = (T_{11}' \to T_{12}')$, hence $T_{12} = T_{12}'$. □

## A.2 Proof of the Order-Isomorphism

We will prove Theorem 3.10, which we restate here:

$\forall E, \Gamma$ : if $FV(E) \subseteq Dom(\Gamma)$, then $(Mig_\Gamma(E), \sqsubseteq)$ and $(Sol(Gen(E, \Gamma)), \leq)$ are order-isomorphic.

*Proof.* If $\varphi$ is a function from type variables to types, then we define the function $G_\varphi$ from terms to terms:

$$\begin{aligned} G_\varphi(\texttt{true}) &= \texttt{true} \\ G_\varphi(\texttt{false}) &= \texttt{false} \\ G_\varphi(n) &= n \\ G_\varphi(x) &= x \\ G_\varphi(\lambda x : T.F) &= \lambda x : \varphi(x).G_\varphi(F) \\ G_\varphi(E_1\ E_2) &= G_\varphi(E_1)\ G_\varphi(E_2) \end{aligned}$$

Let $E, \Gamma$ be given; they remain fixed in the remainder of the proof. Now we define the following function $\alpha_E$ with the help of $G_\varphi$:

$$\begin{aligned} \alpha_E &: Sol(Gen(E,\Gamma)) \to Mig_\Gamma(E) \\ \alpha_E(\varphi) &= G_\varphi(E) \end{aligned}$$

Notice that $\Gamma$ plays no role in the definitions of $G_\varphi$ and $\alpha_E$. We will show that $\alpha_E$ is a well-defined order-isomorphism. We will do this in four steps: we will show that $\alpha_E$ is well defined, injective, and surjective, and that it preserves order.

**Well defined.** We will show that if $\varphi \in Sol(Gen(E,\Gamma))$, then $\alpha_E(\varphi) \in Mig_\Gamma(E)$.

Suppose $\varphi \in Sol(Gen(E,\Gamma))$. We must show

$$E \sqsubseteq \alpha_E(\varphi) \text{ and } \exists T' : \Gamma \vdash \alpha_E(\varphi) : T'.$$

In order to show $E \sqsubseteq \alpha_E(\varphi)$, notice that $E$ and $\alpha_E(\varphi)$ differ only in the type annotations of bound variables. If we have no bound variables in $E$, then $E = \alpha_E(\varphi)$. Otherwise, notice that for every occurrence of $\lambda x : T.F$ in $E$, we have that $\varphi \models T \sqsubseteq x$ and $G_\varphi(\lambda x : T.F) = \lambda x : \varphi(x).G_\varphi(F)$. So, we can show by induction on $E$ that $E \sqsubseteq \alpha_E(\varphi)$.

Define $\mathit{Extend}(\Gamma, E)$ to be $\Gamma$ extended with $(x : T_1)$ for each occurrence in $E$ of $\lambda x : T_1.F$. In order to show $\exists T' : \Gamma \vdash \alpha_E(\varphi) : T'$, we have from Theorem B.3.1 that it is sufficient to prove the stronger statement:

$$\forall E' \text{ subterm of } E : \mathit{Extend}(\Gamma, G_\varphi(E)) \vdash G_\varphi(E') : \varphi([\![E']\!]).$$

We proceed by induction on $E'$.

Case: $E' = \texttt{true}$. Notice that $\varphi \models [\![E']\!] = \texttt{bool}$ and use *T-True*.

Case: $E' = \texttt{false}$. Notice that $\varphi \models [\![E']\!] = \texttt{bool}$ and use *T-False*.

Case: $E' = n$. Notice that $\varphi \models [\![E']\!] = \texttt{int}$ and use *T-Num*.

Case: $E' = x$, where $x$ is free in $E$. Notice that $\varphi \models [\![E']\!] = \Gamma(x)$ and $\mathit{Extend}(\Gamma, G_\varphi(E))(x) = \Gamma(x)$ and use *T-Var*.

Case: $E' = x$, where $x$ is bound in $E$. Notice that $\varphi \models [\![E']\!] = x$ and $\mathit{Extend}(\Gamma, G_\varphi(E))(x) = \varphi(x)$ and use *T-Var*.

Case: $E' = \lambda x : T_1.F$. Notice that $\varphi \models [\![E']\!] = x \to [\![F]\!]$. Notice that $\mathit{Extend}(\Gamma, G_\varphi(E)), (x : T_1) = \mathit{Extend}(\Gamma, G_\varphi(E))$. So, from the induction hypothesis we have $\mathit{Extend}(\Gamma, G_\varphi(E)) \vdash G_\varphi(F) : \varphi([\![F]\!])$. Now we use *T-Abs*.

Case: $E' = E_1\ E_2$. Notice that $\varphi \models [\![E_1]\!] \rhd \langle E_2 \rangle \to [\![E_1\ E_2]\!]$ and $\varphi \models \langle E_2 \rangle \sim [\![E_2]\!]$. From the induction hypothesis we have $\mathit{Extend}(\Gamma, G_\varphi(E)) \vdash G_\varphi(E_1) : \varphi([\![E_1]\!])$ and $\mathit{Extend}(\Gamma, G_\varphi(E)) \vdash G_\varphi(E_2) : \varphi([\![E_2]\!])$. Now we use *T-App*.

**Injective.** We will show that $\alpha_E$ is injective, that is, we will show that

$$\text{if } \alpha_E(\varphi) = \alpha_E(\varphi'), \text{ then } \varphi = \varphi'.$$

Suppose $\alpha_E(\varphi) = \alpha_E(\varphi')$. From the definition of $\alpha_E$ we see that for every occurrence in $E$ of $\lambda x : T_1.F$, we have $\varphi(x) = \varphi'(x)$. We will show that for every occurrence of a subterm $E'$ in $E$, we have $\varphi([\![E']\!]) = \varphi'([\![E']\!])$, and for every occurrence of a subterm $(E_1\ E_2)$ in $E$, we have $\varphi(\langle E_2 \rangle) = \varphi'(\langle E_2 \rangle)$. We proceed by induction on $E'$.

Case: $E' = \texttt{true}$. From $\varphi \models [\![E']\!] = \texttt{bool}$ and $\varphi' \models [\![E']\!] = \texttt{bool}$, we have $\varphi([\![E']\!]) = \texttt{bool} = \varphi'([\![E']\!])$.

Case: $E' = \texttt{false}$. From $\varphi \models [\![E']\!] = \texttt{bool}$ and $\varphi' \models [\![E']\!] = \texttt{bool}$, we have $\varphi([\![E']\!]) = \texttt{bool} = \varphi'([\![E']\!])$.

Case: $E' = n$. From $\varphi \models [\![E']\!] = \texttt{int}$ and $\varphi' \models [\![E']\!] = \texttt{int}$, we have $\varphi([\![E']\!]) = \texttt{int} = \varphi'([\![E']\!])$.

Case: $E' = x$, where $x$ is free in $E$. From $\varphi \models [\![E']\!] = \Gamma(x)$ and $\varphi' \models [\![E']\!] = \Gamma(x)$, we have $\varphi([\![E']\!]) = \Gamma(x) = \varphi'([\![E']\!])$.

Case: $E' = x$, where $x$ is bound in $E$. From $\varphi \models [\![E']\!] = x$ and $\varphi' \models [\![E']\!] = x$, we have $\varphi([\![E']\!]) = \varphi(x) = \varphi'(x) = \varphi'([\![E']\!])$.

Case: $E' = \lambda x : T_1.F$. From the induction hypothesis, we have $\varphi([\![F]\!]) = \varphi'([\![F]\!])$. From $\varphi \models [\![E']\!] = x \to [\![F]\!]$ and $\varphi' \models [\![E']\!] = x \to [\![F]\!]$, we have $\varphi([\![E']\!]) = \varphi(x) \to \varphi([\![F]\!]) = \varphi'(x) \to \varphi'([\![F]\!]) = \varphi'([\![E']\!])$.

Case: $E' = E_1\ E_2$. From the induction hypothesis, we have $\varphi([\![E_1]\!]) = \varphi'([\![E_1]\!])$ and $\varphi([\![E_2]\!]) = \varphi'([\![E_2]\!])$. From $\varphi([\![E_1]\!]) = \varphi'([\![E_1]\!])$ and $\varphi \models [\![E_1]\!] \rhd \langle E_2 \rangle \to [\![E_1\ E_2]\!]$ and $\varphi' \models [\![E_1]\!] \rhd \langle E_2 \rangle \to [\![E_1\ E_2]\!]$, we have $\varphi([\![E']\!]) = \varphi'([\![E']\!])$.

**Surjective.** We will show that $\alpha_E$ is surjective, that is, we will show that

$$\text{if } E_0 \in \mathit{Mig}_\Gamma(E), \text{ then } \exists \varphi \in \mathit{Sol}(\mathit{Gen}(E, \Gamma)) : E_0 = \alpha_E(\varphi).$$

From $E_0 \in \mathit{Mig}_\Gamma(E)$ we have $E \sqsubseteq E_0$ and $T_0$ such that $\Gamma \vdash E_0 : T_0$. From $\Gamma \vdash E_0 : T_0$ and Theorem B.3.1, we have that $\mathit{Extend}(\Gamma, E_0) \vdash E_0 : T_0$.

We define $\varphi$ as follows. Consider a derivation $D$ of $\mathit{Extend}(\Gamma, E_0) \vdash E_0 : T_0$. First, for $x \in \mathit{Dom}(\mathit{Extend}(\Gamma, E_0))$, define $\varphi(x) = \mathit{Extend}(\Gamma, E_0)(x)$. Second, for every occurrence of a subterm $E'$ of $E_0$, find the judgment in $D$ of the form $\Gamma' \vdash E' : T'$, and define $\varphi([\![E']\!]) = T'$. Third, for every occurrence of a subterm $E'$ of the form $E_1\ E_2$ in $E_0$, find the use of *T-App* for $E'$ and in that use, find the condition $T_1 \rhd (T_{11} \to T_{12})$, and define $\varphi(\langle E_2 \rangle) = T_{11}$.

We must show that $\varphi \in \mathit{Sol}(\mathit{Gen}(E, \Gamma))$. We will do a case analysis of the occurrences of subterms $E'$ in $E$.

Case: $E' = \texttt{true}$. From *(T-True)* we have that $\varphi([\![E']\!]) = \texttt{bool}$ so $\varphi \models [\![E']\!] = \texttt{bool}$.

Case: $E' = \texttt{false}$. From *(T-False)* we have that $\varphi([\![E']\!]) = \texttt{bool}$ so $\varphi \models [\![E']\!] = \texttt{bool}$.

Case: $E' = n$. From *(T-Num)* we have that $\varphi([\![E']\!]) = \texttt{int}$ so $\varphi \models [\![E']\!] = \texttt{int}$.

Case: $E' = x$, where $x$ is free in $E$. From *(T-Var)* we have that $\varphi([\![E']\!]) = \varphi(x) = \Gamma(x)$ so $\varphi \models [\![E']\!] = \Gamma(x)$.

Case: $E' = x$, where $x$ is bound in $E$. From *(T-Var)* we have that $\varphi([\![E']\!]) = \varphi(x)$ so $\varphi \models [\![E']\!] = x$.

Case: $E' = \lambda x : T_1.F$. The derivation $D$ contains this use of *T-Abs*:

$$\frac{\Gamma, x : T_1 \vdash F : T_2}{\Gamma \vdash (\lambda x : T_1.F) : T_1 \to T_2}\ \textit{(T-Abs)}$$

So, $\varphi(x) = T_1$ and $\varphi([\![F]\!]) = T_2$ and $\varphi([\![\lambda x : T_1.F]\!]) = T_1 \to T_2$. So, $\varphi \models [\![\lambda x : T_1.F]\!] = x \to [\![F]\!]$. Additionally, we have $E_0 \in Mig_\Gamma(E)$ so if the type annotation of $x$ in $E$ is $S$, then we have $S \sqsubseteq T_1 = \varphi(x)$, so $\varphi \models S \sqsubseteq x$.

Case: $E' = E_1\ E_2$. The derivation $D$ contains this use of *T-App*:

$$\frac{\begin{array}{cc}\Gamma \vdash E_1 : T_1 & \Gamma \vdash E_2 : T_2 \\ T_1 \rhd (T_{11} \to T_{12}) & T_2 \sim T_{11}\end{array}}{\Gamma \vdash E_1\ E_2 : T_{12}}\ \textit{(T-App)}$$

So, $\varphi([\![E_1]\!]) = T_1$ and $\varphi([\![E_2]\!]) = T_2$ and $\varphi([\![E_1\ E_2]\!] = T_{12}$ and $\varphi(\langle E_2 \rangle) = T_{11}$. We have $T_1 \rhd (T_{11} \to T_{12})$. So, $\varphi \models [\![E_1]\!] \rhd \langle E_2 \rangle \to [\![E_1\ E_2]\!]$. Additionally, we have $T_2 \sim T_{11}$. So, $\varphi \models \langle E_2 \rangle \sim [\![E_2]\!]$.

Notice that $\alpha_E(\varphi) = G_\varphi(E) = E_0$. The reason is that $E_0$ differs from $E$ only in the type annotations of bound variables, and for the case of a subterm in $E_0$ of the form $\lambda x : T.F$, we have that $G_\varphi$ replaces the type annotation of $x$ with $\varphi(x) = T$.

**Preserves order.** We will show that $\alpha_E$ preserves order, that is, we will show that

$$\text{if } \varphi \leq \varphi', \text{ then } \alpha_E(\varphi) \sqsubseteq \alpha_E(\varphi').$$

We will prove the following stronger statement:

$$\text{if } \varphi \leq \varphi', \text{ then } \forall E' : G_\varphi(E') \sqsubseteq G_{\varphi'}(E').$$

Suppose that $\varphi \leq \varphi'$. We proceed by induction on $E'$.

Case: $E' = \texttt{true}$. We have $G_\varphi(E') = \texttt{true} = G_{\varphi'}(E')$.

Case: $E' = \texttt{false}$. We have $G_\varphi(E') = \texttt{false} = G_{\varphi'}(E')$.

Case: $E' = n$. We have $G_\varphi(E') = n = G_{\varphi'}(E')$.

Case: $E' = x$. We have $G_\varphi(E') = x = G_{\varphi'}(E')$.

Case: $E' = \lambda x : T.F$. From induction hypothesis, we have $G_\varphi(F) \sqsubseteq G_{\varphi'}(F)$. From $\varphi \leq \varphi'$ we have $\varphi(x) \sqsubseteq \varphi'(x)$. From the definition of $G_\varphi$ and *(P-Abs)* we have $G_\varphi(\lambda x : T.F) = \lambda x : \varphi(x).G_\varphi(F) \sqsubseteq \lambda x : \varphi'(x).G_{\varphi'}(F) = G_{\varphi'}(\lambda x : T.F)$.

Case: $E' = E_1\ E_2$. From induction hypothesis, we have $G_\varphi(E_1) \sqsubseteq G_{\varphi'}(E_1)$ and $G_\varphi(E_2) \sqsubseteq G_{\varphi'}(E_2)$. From the definition of $G_\varphi$ and *(P-App)* we have $G_\varphi(E_1\ E_2) = G_\varphi(E_1)\ G_\varphi(E_2) \sqsubseteq G_{\varphi'}(E_1)\ G_{\varphi'}(E_2) = G_{\varphi'}(E_1\ E_2)$.

The completes the induction proof. We conclude:

$$\alpha_E(\varphi) = G_\varphi(E) \sqsubseteq G_{\varphi'}E = \alpha_E(\varphi')$$

In summary, we have proved all four properties: $\alpha_E$ is well defined, injective, and surjective, and it preserves order. □

## A.3 The Constraint $p \sim (p \to q)$ has no Maximal Solution

**Lemma A.1.** $\{(p, q) \mid p \sim (p \to q)\} = \{(\texttt{Dyn}, b)\} \cup \{(a \to b, c) \mid a \sim (a \to b) \wedge b \sim c\}$.

*Proof.* We will consider each direction in turn.

First consider $\subseteq$. Suppose we have $(p, q)$ such that $p \sim (p \to q)$. Let us divide into four cases of $p$. In one case, we have $p = \texttt{Dyn}$. So, $(p, q) \in \{(\texttt{Dyn}, b)\}$. In another case, we have $p = \texttt{bool}$. However, this is impossible because $p \sim (p \to q)$. In a third case, we have $p = \texttt{int}$. However, this is impossible because $p \sim (p \to q)$. In a fourth case, we have $p = a \to b$, for some $a, b$. We have $(a \to b) \sim ((a \to b) \to q)$, so $a \sim (a \to b)$ and $(b \sim q)$. So, $(p, q) \in \{(a \to b, c) \mid a \sim (a \to b) \wedge b \sim c\}$.

Second consider $\supseteq$. We divide into two subcases. In one case, consider $(\texttt{Dyn}, b)$. We have $\texttt{Dyn} \sim (\texttt{Dyn} \to b)$, so $(\texttt{Dyn}, b) \in (p, q) \mid p \sim (p \to q)$. In another case, consider $(a \to b, c)$, where $a \sim (a \to b) \wedge b \sim c$. We have $(a \to b) \sim ((a \to b) \to c)$, so $(a \to b, c) \in (p, q) \mid p \sim (p \to q)$. □

Define

$$\begin{aligned} \mathit{bump}(\texttt{Dyn}) &= \texttt{Dyn} \to \texttt{Dyn} \\ \mathit{bump}(s \to t) &= \mathit{bump}(s) \to t \end{aligned}$$

**Lemma A.2.** *$\mathit{bump}(a) \sim (\mathit{bump}(a) \to b)$ and $(a \sqsubseteq \mathit{bump}(a))$.*

*Proof.* We proceed by induction on $a$.

In the base case of $a = \texttt{Dyn}$, we have $\mathit{bump}(\texttt{Dyn}) = (\texttt{Dyn} \to \texttt{Dyn}) \sim ((\texttt{Dyn} \to \texttt{Dyn}) \to b) = (\mathit{bump}(a) \to b)$ and $\texttt{Dyn} \sqsubseteq \mathit{bump}(\texttt{Dyn})$ as required.

In the induction step, consider $a = (s \to t)$. The induction hypothesis is that $bump(a) \sim (bump(a) \to b) and (a \sqsubseteq bump(a))$, that is, $bump((s \to t)) \sim (bump((s \to t)) \to b)$ and $((s \to t) \sqsubseteq bump((s \to t)))$. From those properties we get that $(bump(s) \to t) \sim ((bump(s) \to t) \to b)$ and $((s \to t) \sqsubseteq (bump(s) \to t))$. This implies that

$$bump(s) \sim (bump(s) \to t)$$
$$t \sim b$$
$$s \sqsubseteq bump(s)$$

Now let us proceed to proceed to do what we need to do. We have

$$\begin{aligned} & bump(a) \\ = \; & bump(s \to t) \\ = \; & (bump(s) \to t) \\ \sim \; & ((bump(s) \to t) \to b \\ = \; & (bump(s \to t)) \to b \\ = \; & bump(a) \to b \end{aligned}$$

Also,

$$a \;=\; (s \to t) \;\sqsubseteq\; (bump(s)) \to t \;=\; bump(s \to t) \;=\; bump(a)$$

□

**Lemma A.3.** $\{(p, q) \mid p \sim (p \to q)\}$ *has no maximal element.*

*Proof.* Suppose we have $(p, q) \in \{(p, q) \mid p \sim (p \to q)\}$. Our goal is to show that there exists $(c, d)$ such that

$$
\begin{aligned}
&(p, q) \neq (c, d) \ \wedge \\
&(c, d) \in \{(p, q) \mid p \sim (p \to q)\} \ \wedge \\
&p \sqsubseteq c \ \wedge \\
&q \sqsubseteq d
\end{aligned}
$$

We divide into two cases, based on Lemma A.1.

First, suppose $p = \texttt{Dyn}$. We pick $c = \texttt{Dyn} \to \texttt{Dyn}$ and $d = q$. This satisfies the requirements because

$$
\begin{aligned}
&(\texttt{Dyn}, q) \neq ((\texttt{Dyn} \to \texttt{Dyn}), q) \ \wedge \\
&(\texttt{Dyn} \to \texttt{Dyn}) \sim ((\texttt{Dyn} \to \texttt{Dyn}) \to q) \ \wedge \\
&\texttt{Dyn} \sqsubseteq (\texttt{Dyn} \to \texttt{Dyn}) \ \wedge \\
&q \sqsubseteq q.
\end{aligned}
$$

Second, suppose $p = a \to b$. We have $(a \to b) \sim ((a \to b) \to q$ so $a \sim (a \to b)$ and $b \sim q$. We pick $c = bump(a) \to b$ and $d = q$. This satisfies the requirements because of Lemma A.2:

$$
\begin{aligned}
&((a \to b), q) \neq ((bump(a) \to b), q) \ \wedge \\
&(bump(a) \to b) \sim ((bump(a) \to b) \to q) \ \wedge \\
&(a \to b) \sqsubseteq (bump(a) \to b) \ \wedge \\
&q \sqsubseteq q
\end{aligned}
$$

□

## A.4 The Constraints for $\texttt{succ}((\lambda y.y)((\lambda x.x)\texttt{true}))$ have two Maximal Solutions

$$
\begin{aligned}
E &= \texttt{succ}((\lambda y.y)((\lambda x.x)\texttt{true})) \\
\Gamma &= [\ \texttt{succ} : \texttt{int} \to \texttt{int}\ ]
\end{aligned}
$$

First we construct $Gen(E, \Gamma)$:

| | |
|---|---|
| $\texttt{succ}$ | $[\![\texttt{succ}]\!] = \texttt{int} \to \texttt{int}$ |
| $\texttt{succ}((\lambda y.y)((\lambda x.x)\texttt{true}))$ | $[\![\texttt{succ}]\!] \rhd \langle(\lambda y.y)((\lambda x.x)\texttt{true})\rangle \to [\![\texttt{succ}((\lambda y.y)((\lambda x.x)\texttt{true}))]\!]$ |
| | $\langle(\lambda y.y)((\lambda x.x)\texttt{true})\rangle \sim [\![(\lambda y.y)((\lambda x.x)\texttt{true})]\!]$ |
| $\lambda y.y$ | $[\![\lambda y.y]\!] = y \to [\![y]\!]$ |
| | $\texttt{Dyn} \sqsubseteq y$ |
| $y$ | $[\![y]\!] = y$ |
| $(\lambda y.y)((\lambda x.x)\texttt{true})$ | $[\![\lambda y.y]\!] \rhd \langle(\lambda x.x)\texttt{true}\rangle \to [\![(\lambda y.y)((\lambda x.x)\texttt{true})]\!]$ |
| | $\langle(\lambda x.x)\texttt{true}\rangle \sim [\![(\lambda x.x)\texttt{true}]\!]$ |
| $\lambda x.x$ | $[\![\lambda x.x]\!] = x \to [\![x]\!]$ |
| | $\texttt{Dyn} \sqsubseteq x$ |
| $x$ | $[\![x]\!] = x$ |
| $(\lambda x.x)\texttt{true}$ | $[\![\lambda x.x]\!] \rhd \langle\texttt{true}\rangle \to [\![(\lambda x.x)\texttt{true}]\!]$ |
| | $\langle\texttt{true}\rangle \sim [\![\texttt{true}]\!]$ |
| $\texttt{true}$ | $[\![\texttt{true}]\!] = \texttt{bool}$ |

Next we apply *SimPrec* to $Gen(E, \Gamma)$. This step removes $\texttt{Dyn} \sqsubseteq y$ and $\texttt{Dyn} \sqsubseteq x$, which leaves us with the following 12 constraints.

| | |
|---|---|
| $\texttt{succ}$ | $[\![\texttt{succ}]\!] = \texttt{int} \to \texttt{int}$ |
| $\texttt{succ}((\lambda y.y)((\lambda x.x)\texttt{true}))$ | $[\![\texttt{succ}]\!] \rhd \langle(\lambda y.y)((\lambda x.x)\texttt{true})\rangle \to [\![\texttt{succ}((\lambda y.y)((\lambda x.x)\texttt{true}))]\!]$ |
| | $\langle(\lambda y.y)((\lambda x.x)\texttt{true})\rangle \sim [\![(\lambda y.y)((\lambda x.x)\texttt{true})]\!]$ |
| $\lambda y.y$ | $[\![\lambda y.y]\!] = y \to [\![y]\!]$ |
| $y$ | $[\![y]\!] = y$ |
| $(\lambda y.y)((\lambda x.x)\texttt{true})$ | $[\![\lambda y.y]\!] \rhd \langle(\lambda x.x)\texttt{true}\rangle \to [\![(\lambda y.y)((\lambda x.x)\texttt{true})]\!]$ |
| | $\langle(\lambda x.x)\texttt{true}\rangle \sim [\![(\lambda x.x)\texttt{true}]\!]$ |
| $\lambda x.x$ | $[\![\lambda x.x]\!] = x \to [\![x]\!]$ |
| $x$ | $[\![x]\!] = x$ |
| $(\lambda x.x)\texttt{true}$ | $[\![\lambda x.x]\!] \rhd \langle\texttt{true}\rangle \to [\![(\lambda x.x)\texttt{true}]\!]$ |
| | $\langle\texttt{true}\rangle \sim [\![\texttt{true}]\!]$ |
| $\texttt{true}$ | $[\![\texttt{true}]\!] = \texttt{bool}$ |

Let us use $A_{12}$ to denote the above set of 12 constraints. In the listing $A_{12}$, we have three Matching constraints, which for brevity of notation, we will number from 1 to 3, as follows:

$$1 \;:\; [\![\texttt{succ}]\!] \rhd \langle(\lambda y.y)((\lambda x.x)\texttt{true})\rangle \to [\![\texttt{succ}((\lambda y.y)((\lambda x.x)\texttt{true}))]\!]$$

$$2 \;:\; [\![\lambda y.y]\!] \rhd \langle(\lambda x.x)\texttt{true}\rangle \to [\![(\lambda y.y)((\lambda x.x)\texttt{true})]\!]$$

$$3 \;:\; [\![\lambda x.x]\!] \rhd \langle\texttt{true}\rangle \to [\![(\lambda x.x)\texttt{true}]\!]$$

Now we must consider all subsets of $\{1, 2, 3\}$. For each $S \subseteq \{1, 2, 3\}$, we must determine whether $SimMatch(A_{12}, S)$ has finitely many solutions.

We can see easily that we must focus on $S = \{1, 2, 3\}$; for any of the other choices of $S$, we have that $SimMatch(A_{12}, S)$ is unsatisfiable. So, we construct $SimMatch(A_{12}, \{1, 2, 3\})$:

| | |
|---|---|
| $\texttt{succ}$ | $[\![\texttt{succ}]\!] = \texttt{int} \to \texttt{int}$ |
| $\texttt{succ}((\lambda y.y)((\lambda x.x)\texttt{true}))$ | $[\![\texttt{succ}]\!] = \langle(\lambda y.y)((\lambda x.x)\texttt{true})\rangle \to [\![\texttt{succ}((\lambda y.y)((\lambda x.x)\texttt{true}))]\!]$ |
| | $\langle(\lambda y.y)((\lambda x.x)\texttt{true})\rangle \sim [\![(\lambda y.y)((\lambda x.x)\texttt{true})]\!]$ |
| $\lambda y.y$ | $[\![\lambda y.y]\!] = y \to [\![y]\!]$ |
| $y$ | $[\![y]\!] = y$ |
| $(\lambda y.y)((\lambda x.x)\texttt{true})$ | $[\![\lambda y.y]\!] = \langle(\lambda x.x)\texttt{true}\rangle \to [\![(\lambda y.y)((\lambda x.x)\texttt{true})]\!]$ |
| | $\langle(\lambda x.x)\texttt{true}\rangle \sim [\![(\lambda x.x)\texttt{true}]\!]$ |
| $\lambda x.x$ | $[\![\lambda x.x]\!] = x \to [\![x]\!]$ |
| $x$ | $[\![x]\!] = x$ |
| $(\lambda x.x)\texttt{true}$ | $[\![\lambda x.x]\!] = \langle\texttt{true}\rangle \to [\![(\lambda x.x)\texttt{true}]\!]$ |
| | $\langle\texttt{true}\rangle \sim [\![\texttt{true}]\!]$ |
| $\texttt{true}$ | $[\![\texttt{true}]\!] = \texttt{bool}$ |

Next we apply *SimEq* to $SimMatch(A_{12}, \{1,2,3\})$. Notice that $SimMatch(A_{12}, \{1,2,3\})$ has 9 Equality constraints. Those 9 Equality constraints are satisfiable and have the following most general unifier ($\varphi_{123}$), where $p, q$ are type variables:

| $v$ | : | $\varphi_{123}(v)$ |
|---|---|---|
| $[\![\texttt{succ}]\!]$ | : | $\texttt{int} \to \texttt{int}$ |
| $[\![\texttt{succ}((\lambda y.y)((\lambda x.x)\texttt{true}))]\!]$ | : | $\texttt{int}$ |
| $\langle(\lambda y.y)((\lambda x.x)\texttt{true})\rangle$ | : | $\texttt{int}$ |
| $[\![(\lambda y.y)((\lambda x.x)\texttt{true})]\!]$ | : | $q$ |
| $[\![\lambda y.y]\!]$ | : | $q \to q$ |
| $y$ | : | $q$ |
| $[\![y]\!]$ | : | $q$ |
| $\langle(\lambda x.x)\texttt{true}\rangle$ | : | $q$ |
| $[\![(\lambda x.x)\texttt{true}]\!]$ | : | $p$ |
| $[\![\lambda x.x]\!]$ | : | $p \to p$ |
| $x$ | : | $p$ |
| $[\![x]\!]$ | : | $p$ |
| $\langle\texttt{true}\rangle$ | : | $p$ |
| $[\![\texttt{true}]\!]$ | : | $\texttt{bool}$ |

Let us use $A'$ to denote the subset of 3 Consistency constraints in $SimMatch(A_{12}, \{1,2,3\})$, which is:

$$\begin{aligned}
\langle(\lambda y.y)((\lambda x.x)\texttt{true})\rangle &\sim [\![(\lambda y.y)((\lambda x.x)\texttt{true})]\!] \\
\langle(\lambda x.x)\texttt{true}\rangle &\sim [\![(\lambda x.x)\texttt{true}]\!] \\
\langle\texttt{true}\rangle &\sim [\![\texttt{true}]\!]
\end{aligned}$$

Next we apply $\varphi_{123}$ to $A'$. The result is that $\mathit{SimEq}(\mathit{SimMatch}(A_{12}, \{1,2,3\}), \varphi_{123})$ is:

$$\begin{array}{rcl} \texttt{int} & \sim & q \\ q & \sim & p \\ p & \sim & \texttt{bool} \end{array}$$

Let us use $A_{123}$ to denote the above set of 3 Consistency constraints. Next we apply *SimCon* to $A_{123}$. The effect is to change $\texttt{int} \sim q$ into $q \sim \texttt{int}$:

$$\begin{array}{rcl} q & \sim & \texttt{int} \\ q & \sim & p \\ p & \sim & \texttt{bool} \end{array}$$

Let us use $A_{cm}$ to denote the above set of 3 Consistency constraints. We observe that $\mathit{Bounded}(A_{cm})$. Now we use Theorem B.4.6 to conclude that $\mathit{Sol}(A_{cm})$ is finite.

Notice that $\mathit{SimCon}(A_{123})$ has three solutions:

| | | |
|---|---|---|
| $\varphi_1$ | $x$ : Dyn; | $y$ : Dyn |
| $\varphi_2$ | $x$ : Dyn; | $y$ : int |
| $\varphi_3$ | $x$ : bool; | $y$ : Dyn |

Notice that each of $\varphi_2$ and $\varphi_3$ is a maximal solution, but neither is a greatest solution.

# APPENDIX B

# Design and Migration of Gradual Rank-2 Intersection Types

## B.1 The Rank-2 intersections $\lambda$-calculus

We present the Static Rank-2 $\lambda$-calculus in Figure B.1.

**Definition B.1** (Concretization)**.** Let $\gamma : T \to \mathcal{P}(T)$ be defined as follows:

$\gamma(\texttt{Dyn}) = T^2 \qquad \gamma(\texttt{Int}) = \{\texttt{Int}\} \qquad \gamma(\texttt{Bool}) = \{\texttt{Bool}\}$

$\gamma(T_1^0 \to T_2^0) = \{\tau_1 \to \tau_2 \mid \tau_i \in \gamma(T_i^0)\} \qquad \gamma(T_1^1 \wedge T_2^1) = \{\tau_1 \wedge \tau_2 \mid \tau_i \in \gamma(T_i^1)\}$

$\gamma(T_1^1 \to T_2^2) = \{\tau_1 \to \tau_2 \mid \tau_1 \in \gamma(T_1^1) \wedge \tau_2 \in \gamma(T_2^2)\}$

**Lemma B.1.** *(Type precision) $T \sqsubseteq T'$ if and only if $\gamma(T) \subseteq \gamma(T')$.*

*Proof.* Straightforward by induction. □

### B.1.1 Proof of conservative extension

**Theorem 4.7.4** (Conservative Extension of Static Rank-2)**.** *For all static $\Gamma, E$ and $T$, we have:*

*1. $\Gamma \vdash_S E : T$ iff $\Gamma \vdash_f E : T$.*

*2. $E \Downarrow_S v$ iff $E \Downarrow_f v$*

$$
\begin{array}{rrcl}
\text{(Types)} & \tau^0, \sigma^0, T^0 & ::= & \texttt{Bool} \mid \texttt{Int} \mid T^0 \to T^0 \\
& \tau^1, \sigma^1, T^1 & ::= & T^0 \mid T^1 \wedge T^1 \\
& \tau^2, \sigma^2, T^2 & ::= & T^0 \mid T^1 \to T^2 \\
& \tau, \sigma, T & ::= & T^0 \mid T^1 \mid T^2 \\
\text{(Terms)} & E & ::= & \texttt{true} \mid \texttt{false} \mid n \mid x_l \mid \lambda x : \sigma^1.E \mid E\ E \\
\text{(Environments)} & \Gamma & ::= & \emptyset \mid \Gamma, x : \sigma^1 \\
\text{(Labels)} & & ::= & l \in \mathbb{N}
\end{array}
$$

Typing Rules:

$\Gamma \vdash_S E : T^2.$

$\Gamma \vdash_S \texttt{true} : \texttt{Bool}$ *(T-True)* $\quad$ $\Gamma \vdash_S \texttt{false} : \texttt{Bool}$ *(T-False)* $\quad$ $\Gamma \vdash_S n : \texttt{Int}$ *(T-Num)*

$$\frac{x : \bigwedge_{i \in I} \tau_i^0 \in \Gamma \quad l \in I}{\Gamma \vdash_S x_l : \tau_l^0}\ (T\text{-}Var)$$

$$\frac{\begin{array}{c}\Gamma, (x : \sigma_0^0 \wedge \ldots \wedge \sigma_n^0) \vdash_S E : \tau \\ \textit{where } \forall i \in \{1, \ldots, n\}, x_i \textit{ occurs in } E \\ \textit{and } \sigma_0^0 = \ldots = \sigma_n^0 = \sigma^0\end{array}}{\Gamma \vdash_S (\lambda x : \sigma^0.E) : \sigma^0 \to \tau}\ \textit{(T-Abs-0)}$$

$$\frac{\Gamma, (x : \sigma_0^0 \wedge ... \wedge \sigma_n^0) \vdash_S E : \tau \quad \textit{where } \forall i \in \{1, ..., n\},\ x_i \textit{ occurs in } E}{\Gamma \vdash_S (\lambda x : \sigma_0^0 \wedge ... \wedge \sigma_n^0.E) : \sigma_0^0 \wedge ... \wedge \sigma_n^0 \to \tau}\ \textit{(T-Abs-1)}$$

$$\frac{\Gamma \vdash_S E : \tau \quad \textit{where } x_0 \textit{ does not occur in } E}{\Gamma \vdash_S (\lambda x : \tau^0.E) : \tau^0 \to \tau}\ \textit{(T-Abs-2)}$$

$$\frac{\begin{array}{cc}\Gamma \vdash_S E_1 : \sigma_1 & \Gamma \vdash_S E_2 : \sigma^0 \\ \sigma_1 = ((\bigwedge_{i \in I} \sigma^0) \to \sigma) & \end{array}}{\Gamma \vdash_S E_1\ E_2 : \sigma}\ \textit{(T-App)}$$

**Figure B.1:** The Rank-2 intersections $\lambda$-calculus.

*Proof.* Proof of item 1.

Cases $x, n, True, False, \lambda x : T.E$: Straightforward, since the typing rules are the same in

both cases.

Case $E_1\ E_2$: Forward direction:

$$\frac{\begin{array}{cc}\Gamma \vdash_S E_1 : \sigma_1 & \Gamma \vdash_S E_2 : \sigma^0 \\ \sigma_1 = ((\bigwedge_{i \in I} \sigma^0) \to \sigma) & \end{array}}{\Gamma \vdash_S E_1\ E_2 : \sigma} \textit{(T-App)}$$

From the IH we get:

$$\frac{\begin{array}{cc}\Gamma \vdash E_1 : \sigma_1 & \Gamma \vdash E_2 : \sigma^0 \\ \sigma_1 \rhd ((\bigwedge_{i \in I} \tau_i) \to \sigma) & \forall i \leq I : \sigma^0 \sim \tau_i \end{array}}{\Gamma \vdash E_1\ E_2 : \sigma} \textit{(T-App)}$$

Backward direction:

$$\frac{\begin{array}{cc}\Gamma \vdash E_1 : \sigma_1 & \Gamma \vdash E_2 : \sigma^0 \\ \sigma_1 \rhd ((\bigwedge_{i \in I} \tau_i) \to \sigma) & \forall i \leq I : \sigma^0 \sim \tau_i \end{array}}{\Gamma \vdash E_1\ E_2 : \sigma} \textit{(T-App)}$$

From the IH we have that $\Gamma \vdash_S E_1 : \sigma_1$ and $\Gamma \vdash_S E_2 : \sigma_0$. Since $\Gamma, E$ and $T$ are static, we know that $\sigma_1 = ((\bigwedge_{i \in I} \tau_i) \to \sigma)$ and $\forall i \leq I : \sigma^0 = \tau_i$ so $\Gamma \vdash_S E_1\ E_2 : \sigma$ and we are done

Proof of item 2 is straightforward. □

### B.1.2 Proof of the Unique Types theorem

**Theorem 4.4.1** (Unique Type)**.** $\forall \Gamma, E, T, T'$, *if* $\Gamma \vdash E : T$ *and* $\Gamma \vdash E : T'$, *then* $T = T'$.

*Proof.* We proceed by induction on the derivation of $\Gamma \vdash E : T$. We will do a case analysis based on the last rule that was used to derive $\emptyset \vdash E : T$.

Case: *T-Num*. The last rule that was used to derive $\emptyset \vdash E : T'$ must be *T-Num*, so $T = \texttt{Int} = T'$.

Case: *T-True*. The last rule that was used to derive $\emptyset \vdash E : T'$ must be *T-True*, so $T = \texttt{Bool} = T'$.

Case: *T-False*. The last rule that was used to derive $\emptyset \vdash E : T'$ must be *T-False*, so $T = \texttt{Bool} = T'$.

Case: *T-Var*. We have that the rule $x : \tau^0 \vdash E : \tau^0$ must be *T-Var*, so $T = \Gamma(x) = T'$.

Case: *T-Abs-0*. The last step of the derivation of $\Gamma \vdash E : T$ must be as follows:

$$\frac{\begin{array}{c}\Gamma, (x : \sigma_0^0 \wedge \ldots \wedge \sigma_n^0) \vdash_S E : \tau \\ \textit{where } \forall i \in \{1, \ldots, n\}, x_i \textit{ occurs in } E \\ \textit{and } \sigma_0^0 = \ldots = \sigma_n^0 = \sigma^0\end{array}}{\Gamma \vdash_S (\lambda x : \sigma^0.E) : \sigma^0 \to \tau} \textit{ (T-Abs-0)}$$

The last rule that was used to derive $\Gamma \vdash E : T'$ must be *T-Abs-0*, as follows:

$$\frac{\begin{array}{c}\Gamma, (x : \sigma_0^0 \wedge \ldots \wedge \sigma_n^0) \vdash E : \tau' \\ \textit{where } \forall i \in \{1, \ldots, n\}, x_i \textit{ occurs in } E \\ \textit{and } \sigma_0^0 = \ldots = \sigma_n^0 = \sigma^0\end{array}}{\Gamma \vdash (\lambda x : \sigma^0.E) : \sigma^0 \to \tau'} \textit{ (T-Abs-0)}$$

For the subderivations $\Gamma, (x : \sigma^0 \wedge ... \wedge \sigma^0) \vdash F : \tau$ and $\Gamma, (x : \sigma^0 \wedge ... \wedge \sigma^0) \vdash F : \tau'$, we apply the Induction Hypothesis to get that $\tau = \tau'$. So we are done.

Case: *T-Abs-1.* The last step of the derivation of $\Gamma \vdash E : T$ must be as follows:

$$\frac{\Gamma, (x : \sigma_0^0 \wedge ... \wedge \sigma_n^0) \vdash E : \tau \quad \textit{where } \forall i \in \{1, ..., n\},\ x_i \textit{ occurs in } E}{\Gamma \vdash (\lambda x : \sigma_0^0 \wedge ... \wedge \sigma_n^0 . E) : \sigma_0^0 \wedge ... \wedge \sigma_n^0 \to \tau} \textit{ (T-Abs-1)}$$

The last rule that was used to derive $\Gamma \vdash E : T'$ must be *T-Abs-1*, as follows:

$$\frac{\Gamma, (x : \sigma_0^0 \wedge ... \wedge \sigma_n^0) \vdash E : \tau' \quad \textit{where } \forall i \in \{1, ..., n\},\ x_i \textit{ occurs in } E}{\Gamma \vdash (\lambda x : \sigma_0^0 \wedge ... \wedge \sigma_n^0 . E) : \sigma_0^0 \wedge ... \wedge \sigma_n^0 \to \tau'} \textit{ (T-Abs-1)}$$

For the subderivations $\Gamma, (x : \sigma_0^0 \wedge ... \wedge \sigma_n^0) \vdash F : \tau$ and $\Gamma, (x : \sigma_0^0 \wedge ... \wedge \sigma_n^0) \vdash F : \tau'$, we apply the Induction Hypothesis to get that $\tau = \tau'$. So we are done.

Case: *T-Abs-2* The last step of the derivation of $\Gamma \vdash E : T$ must be as follows:

$$\frac{\Gamma \vdash F : \tau \quad \textit{where } x \textit{ does not occur in } F}{\Gamma \vdash (\lambda x : \tau^0 . E) : \tau^0 \to \tau} \textit{ (T-Abs-2)}$$

The last rule that was used to derive $\Gamma \vdash E : T'$ must be *T-Abs-2*, as follows:

$$\frac{\Gamma \vdash F : \tau' \quad \textit{where } x \textit{ does not occur in } F}{\Gamma \vdash (\lambda x : \tau^0 . E) : \tau^0 \to \tau'} \textit{ (T-Abs-2)}$$

For the subderivations $\Gamma \vdash F : \tau$ and $\Gamma \vdash F : \tau'$, we apply the Induction Hypothesis to get that $\tau = \tau'$. So we are done.

Case: *T-App.* The last step of the derivation of $\Gamma \vdash E : \sigma$ must be as follows:

$$\frac{\begin{array}{cc} \Gamma \vdash E_1 : \sigma_1 & \Gamma \vdash E_2 : \sigma^0 \\ \sigma_1 \rhd ((\bigwedge_{i \in I} \tau_i) \to \sigma) & \forall i \leq I : \sigma^0 \sim \tau_i \end{array}}{\Gamma \vdash E_1\ E_2 : \sigma} \textit{ (T-App)}$$

The last rule that was used to derive $\Gamma \vdash E : \sigma'$ must be *T-App*, as follows:

$$\frac{\begin{array}{cc}\Gamma \vdash E_1 : \sigma_1' & \Gamma \vdash E_2 : \sigma^{0'} \\ \sigma_1' \rhd ((\bigwedge_{i \in I} \tau_i') \to \sigma') & \forall i \leq I' : \sigma^{0'} \sim \tau_{i'}\end{array}}{\Gamma \vdash E_1\ E_2 : \sigma'}\ \textit{(T-App)}$$

For the subderivations $\Gamma \vdash E_1 : \sigma_1$ and $\Gamma \vdash E_1 : \sigma_1'$, we apply the Induction Hypothesis to get that $\sigma_1 = \sigma_1'$. Similarly, for the subderivations $\Gamma \vdash E_2 : \sigma^0$ and $\Gamma \vdash E_2 : \sigma^{0'}$, we apply the Induction Hypothesis to get that $\sigma^0 = \sigma^{0'}$. From $\sigma_1 = \sigma_1'$ and $\sigma_1 \rhd ((\bigwedge_{i \in I} \tau_i) \to \sigma')$ and $\tau_1' \rhd ((\bigwedge_{i \in I} \tau_i') \to \sigma')$, we conclude that $(\bigwedge_{i \in I} \tau_i) \to \sigma' = (\bigwedge_{i \in I} \tau_i') \to \sigma'$, hence $\sigma = \sigma'$. □

## B.2 Criteria of Rank-2

**Theorem 4.4.9** (Gradual Guarantee)**.** $\forall \Gamma, \Gamma', E, E', T$ *suppose* $\Gamma \vdash E : T$ *and* $\Gamma' \sqsubseteq \Gamma$ *and* $E' \sqsubseteq E$.

1. *For some* $T'$*, we have* $\Gamma' \vdash E' : T'$ *and* $T' \sqsubseteq T$.
2. *If* $E \Downarrow v$ *then* $E' \Downarrow v'$ *and* $v' \sqsubseteq v$
3. *If* $E \Uparrow$ *then* $E' \Uparrow$
4. *If* $E' \Downarrow v'$ *then* $E \Downarrow v$ *where* $v' \sqsubseteq v$ *or* $E \Downarrow blame_{\mathcal{T}(T)}\mathcal{L}$
5. *If* $E' \Uparrow$ *then* $E \Uparrow$ *or* $E \Downarrow blame_{\mathcal{T}(T)}\mathcal{L}$.

*Proof.* Consider item 1. For the first statement, we proceed by induction. We replicate the proof technique from [CS16].

Proof by induction on the proof structure of $E' \sqsubseteq E$.

Case: $n \sqsubseteq n$. This case is straightforward.

Case: $True \sqsubseteq True$. This case is straightforward.

Case: $False \sqsubseteq False$. This case is straightforward.

Case $x \sqsubseteq x$. This case is straightforward.

case: $E_1'\ E_2' \sqsubseteq E_1\ E_2$ Then by inspection on how $E_1'\ E_2' \sqsubseteq E_1\ E_2$ holds, we have that $E_1' \sqsubseteq E_1$ and $E_2' \sqsubseteq E_2$. We also have the following type rule:

$$\frac{\begin{array}{cc}\Gamma \vdash E_1 : \sigma_1 & \Gamma \vdash E_2 : \sigma^0 \\ \sigma_1 \rhd ((\bigwedge_{i \in I} \tau_i) \to \sigma) & \forall i \leq I : \sigma^0 \sim \tau_i\end{array}}{\Gamma \vdash E_1\ E_2 : \sigma}\ \textit{(T-App)}$$

We need to prove that $\exists \sigma'$ s.t. $\Gamma' \vdash E_1' E_2' : \sigma'$ and $\sigma' \sqsubseteq \sigma$ where $\Gamma' \sqsubseteq \Gamma'$

Since $E_1' \sqsubseteq E_1$ (with smaller proof than $E_1' E_2' \sqsubseteq E_1 E_2$) and $\Gamma \vdash E_1 : \sigma_1$ and $\Gamma' \sqsubseteq \Gamma$, we can apply the IH and obtain $\Gamma' \vdash E_1' : \sigma_1'$ with $\sigma_1' \sqsubseteq \sigma_1$, for some $\sigma_1'$. Since $E_2' \sqsubseteq E_2$ (with smaller proof than $E_1' E_2' \sqsubseteq E_1 E_2$) and $\Gamma \vdash E_2 : \sigma^0$, we can apply the IH and obtain $\Gamma' \vdash E_2' : \sigma^{0'}$ with $\sigma^{0'} \sqsubseteq \sigma^0$.

Our general goal now is to apply *T-App* to prove $\Gamma' \vdash E_1' E_2' : \sigma'$, for some $\sigma'$. Now we show that $\sigma_1 \rhd ((\bigwedge_{i \in I} \tau_i) \to \sigma)$ and $\sigma^0 \sim \tau_i$

From Lemma B.2, since $\sigma_1' \sqsubseteq \sigma_1$ and $\sigma_1 \rhd ((\bigwedge_{i \in I} \tau_i) \to \sigma$, we have $\sigma_1' \rhd ((\bigwedge_{i \in I} \tau_i' \to \sigma_1')$, for some $\tau_i'.\sigma_1'$ where $\tau_i' \sqsubseteq \tau_i$ and $\sigma_1' \sqsubseteq \sigma_1$ for all $i \in I$. So we get that $\sigma_1' \rhd ((\bigwedge_{i \in I} \tau_i' \to \sigma_1')$ can be used in $T - APP$.

From Lemma B.3, since $\sigma_0 \sim \tau_i$ for all $i \in I$ and $\sigma_0' \sqsubseteq \sigma_0$ and $\tau_i' \sqsubseteq \tau_i$, we obtain $\sigma_0' \sim \tau_i'$. Therefore, we can use $T - APP$ in the following way:

$$\frac{\begin{array}{cc}\Gamma' \vdash E'_1 : \sigma'_1 & \Gamma' \vdash E'_2 : \sigma^{0'} \\ \sigma'_1 \rhd ((\bigwedge_{i \in I} \tau'_i) \to \sigma') & \forall i \leq I : \sigma^{0'} \sim \tau_{i'}\end{array}}{\Gamma' \vdash E'_1\ E'_2 : \sigma'} \textit{ (T-App)}$$

As we have established earlier that $\sigma' \sqsubseteq \sigma$ so we are done.

case: $\lambda x : \tau^{0'}.E' \sqsubseteq \lambda x : \tau^0.E$ where $x$ does not occur in $E$

Then by inspection on how $\lambda x : \tau^{0'}.E' \sqsubseteq \lambda x : \tau^0.E$ holds, we have $\tau^{0'} \sqsubseteq \tau^{0'}$ and $E' \sqsubseteq E$. So we can use *T-Abs-2* in the following way

$$\frac{\Gamma \vdash E : \tau \ \ \textit{where } x \textit{ does not occur in } E}{\Gamma \vdash (\lambda x : \tau^0.E) : \tau^0 \to \tau} \textit{ (T-Abs-2)}$$

Since $E' \sqsubseteq E$ (with smaller proof than $\lambda x : \tau^{0'}.E' \sqsubseteq \lambda x : \tau^0.E$ and $\Gamma', x : \tau^{0'} \sqsubseteq \Gamma, x : \tau^0$ and $\Gamma \vdash E : \tau$, we can apply the IH and obtain $\Gamma' \vdash E' : \tau'$, for some $\tau'$ s.t. $\tau' \sqsubseteq \tau$. Therefore we can use *T-Abs-2* in the following way

$$\frac{\Gamma' \vdash E' : \tau' \ \ \textit{where } x \textit{ does not occur in } E'}{\Gamma' \vdash (\lambda x : \tau^{0'}.E') : \tau^{0'} \to \tau'} \textit{ (T-Abs-2)}$$

Now, we need to prove that $\tau^{0'} \to \tau' \sqsubseteq \tau^0 \to \tau$. By definition of $\sqsubseteq$ as a precongruence, $\tau^{0'} \sqsubseteq \tau^0$ (which we have) and $\tau' \sqsubseteq \tau'$ (which we have) imply $\tau^{0'} \to \tau' \sqsubseteq \tau^{0'} \to \tau$. And we are done.

Case $\lambda x : \mathit{Dyn}.E \sqsubseteq \lambda x : \sigma_1^{0'} \wedge ... \wedge \sigma_n^{0'}.E'$ where $x$ occurs in $E$

We can use *T-Abs-1* in the following way:

$$\frac{\Gamma, (x : \sigma_0^0 \wedge ... \wedge \sigma_n^0) \vdash E : \tau \;\; \textit{where } \forall i \in \{1, ..., n\},\ x_i \textit{ occurs in } E}{\Gamma \vdash (\lambda x : \sigma_0^0 \wedge ... \wedge \sigma_n^0.E) : \sigma_0^0 \wedge ... \wedge \sigma_n^0 \to \tau} \textit{ (T-Abs-1)}$$

From precision we have that $\texttt{Dyn} \sqsubseteq \bigwedge_0^n \sigma_i^0$

If we recall the definition of precision on type environments, we can obtain $\Gamma', (x : \texttt{Dyn} \wedge ... \wedge \texttt{Dyn}) \sqsubseteq \Gamma, (x : \sigma_0^0 \wedge ... \wedge x : \sigma_n^0)$.

Since $E' \sqsubseteq E$ (with smaller proof than $\lambda x : \texttt{Dyn}.E' \sqsubseteq \lambda x : \sigma_0^0 \wedge ... \wedge \sigma_n^0.E$ and $\Gamma', x : \texttt{Dyn} \wedge ... \wedge \texttt{Dyn} \sqsubseteq \Gamma, x : \sigma_0^0 \wedge ... \wedge \sigma_n^0$ and $\Gamma, (x : \sigma_0^0 \wedge ... \wedge \sigma_n^0) \vdash E : \tau$, we can apply the IH and obtain $\Gamma', (x : \texttt{Dyn} \wedge ... \wedge \texttt{Dyn}) \vdash E' : \tau'$, for some $\tau'$ s.t. $\tau' \sqsubseteq \tau$. Therefore we can use *T-Abs-0* in the following way:

$$\frac{\Gamma', (x : \texttt{Dyn} \wedge ... \wedge \texttt{Dyn}) \vdash E' : \tau' \;\; \textit{where } \forall i \in \{1, ..., n\},\ x_i \textit{ occurs in } E}{\Gamma' \vdash (\lambda x : \texttt{Dyn}.E') : \texttt{Dyn} \to \tau'} \textit{ (T-Abs-0)}$$

Case $\lambda x : \sigma_0^0 \wedge ... \wedge \sigma_n^0.E \sqsubseteq \lambda x : \sigma_1^{0'} \wedge ... \wedge \sigma_n^{0'}.E'$ where $x$ occurs in $E$

We can use *T-Abs-1* in the following way:

$$\frac{\Gamma, (x : \sigma_0^0 \wedge ... \wedge \sigma_n^0) \vdash E : \tau \;\; \textit{where } \forall i \in \{1, ..., n\},\ x_i \textit{ occurs in } E}{\Gamma \vdash (\lambda x : \sigma_0^0 \wedge ... \wedge \sigma_n^0.E) : \sigma_0^0 \wedge ... \wedge \sigma_n^0 \to \tau} \textit{ (T-Abs-1)}$$

From precision we have that $\forall i : \sigma_i^{0'} \sqsubseteq \sigma_i^0$

If we recall the definition of precision on type environments, we can obtain $\Gamma', (x : \sigma_1^{0'} \wedge ... \wedge \sigma_n^{0'}) \sqsubseteq \Gamma, (x : \sigma_0^0 \wedge ... \wedge \sigma_n^0)$, because $\Gamma' \sqsubseteq \Gamma$ and $\forall i \in \{1, ..., n\} : \sigma_i^{0'} \sqsubseteq \sigma_i^0$.

Since $E' \sqsubseteq E$ (with smaller proof than $\lambda x : \sigma^{0'} \wedge ... \wedge \sigma_n^0.E' \sqsubseteq \lambda x : \sigma_0^0 \wedge ... \wedge \sigma_n^0.E$ and $\Gamma', x : \sigma_1^{0'} \wedge ... \wedge \sigma_n^{0'} \sqsubseteq \Gamma, x : \sigma_0^0 \wedge ... \wedge \sigma_n^0$ and $\Gamma, (x : \sigma_0^0), ..., (x : \sigma_n^0) \vdash E : \tau$, we can apply the IH and obtain $\Gamma', (x\sigma_1^{0'} \wedge ... \wedge \sigma_n^{0'}) \vdash E' : \tau'$, for some $\tau'$ s.t. $\tau' \sqsubseteq \tau$. Therefore we can use $T - Abs$ in the following way

$$\frac{\Gamma', (x : \sigma_1^{0'} \wedge ... \wedge \sigma_n^{0'}) \vdash E' : \tau' \;\; \textit{where } \forall i \in \{1, ..., n\},\ x_i \textit{ occurs in } E}{\Gamma' \vdash (\lambda x : \sigma_1^{0'} \wedge ... \wedge \sigma_n^{0'}.E) : \sigma_1^{0'} \wedge ... \wedge \sigma_n^{0'} \to \tau'} \textit{ (T-Abs-1)}$$

Now, we need to prove that $\sigma^{0'} \wedge ... \wedge \sigma^{n'} \to \tau' \sqsubseteq \tau_1^0 \wedge ... \wedge \tau_n^0 \to \tau$. By definition of $\sqsubseteq$ as a precongruence, we are done. We know that $\tau' \sqsubseteq \tau$ and that $\sigma^{0'} \wedge ... \wedge \sigma^{n'} \to \tau' \sqsubseteq \sigma_0^0 \wedge ... \wedge \sigma_n^0 \to \tau$ so we are done.

Case $\lambda x : \sigma^0.E \sqsubseteq \lambda x : \sigma^{0'}.E'$ where $x$ occurs in $E$

We can use *T-Abs-0* in the following way:

$$\frac{\Gamma, (x : \sigma^0 \wedge ... \wedge \sigma^0) \vdash E : \tau \;\; \textit{where } \forall i \in \{1, ..., n\},\ x_i \textit{ occurs in } E}{\Gamma \vdash (\lambda x : \sigma^0.E) : \sigma^0 \to \tau} \textit{ (T-Abs-0)}$$

From precision, we have that $\sigma^{0'} \sqsubseteq \sigma^0$.

Since $E' \sqsubseteq E$, we can apply the IH and obtain that $\Gamma', (x : \sigma^0 \wedge ... \wedge \sigma^0) : E' : t'$ for some $t' \sqsubseteq t$. So we can use *T-Abs-0* in the following way:

$$\frac{\Gamma', (x_1 : \sigma^{0'} \wedge ... \wedge \sigma^{0'}) \vdash E' : \tau' \;\; \textit{where } \forall i \in \{1, ..., n\},\ x_i \textit{ occurs in } E}{\Gamma \vdash (\lambda x : \sigma^{0'}.E') : \sigma^{0'} \to \tau} \; \textit{(T-Abs-0)}$$

And we have that $\sigma^{0'} \to t' \sqsubseteq \sigma^0 \to t$ since $\sigma^{0'} \sqsubseteq \sigma^0$ and $t \sqsubseteq t$ so we are done.

For the remaining items, first note that $\mathcal{T}$ preserves typability. Therefore, we can directly apply the proof from [SVC15a] on GTLC for items 3 and 4. For 2 and 5 we also use the fact that $\mathcal{T}$ is monotone on the precision ordering. □

**Lemma B.2** (Matching is monotone in its first argument.)**.** *if* $T \triangleright \tau_1 \to \tau_2$ *and* $T' \sqsubseteq T$ *then there exists* $\tau_1', \tau_2'$ *s.t.* $T' \triangleright \tau_1' \to \tau_2'$ *and* $\tau_1' \sqsubseteq \tau_1$ *and* $\tau_2' \sqsubseteq \tau_2$.

**Lemma B.3** (consistency is monotone.)**.** *If* $\tau_1 \sim \tau_2$ *and* $\tau_1' \sqsubseteq \tau_1$ *and* $\tau_{2'} \sqsubseteq \tau_2$ *then* $\tau_{1'} \sim \tau_{2'}$.

*Proof.* We will proceed induction on the structure of $\tau_1 \sim \tau_2$.

Suppose we have $\tau_1 \sim \texttt{Dyn}$. Then $\tau_2' = \texttt{Dyn}$ so by $C - Dyn2$, we are done. A similar argument applies to the case of $\texttt{Dyn} \sim \tau_2$.

Case $\texttt{Bool} \sim \texttt{Bool}$. We have $\tau_1' - \tau_2' = \texttt{Bool}$ so we are done. The same applies for $\texttt{Int} \sim \texttt{Int}$.

Case $(\sigma \to \tau) \sim (\sigma' \to \tau')$. We have that $\sigma \sim \sigma'$ and $\tau \sim \tau'$. Then we have $s \sqsubseteq \sigma \to \tau$ and $u \sqsubseteq \sigma' \to \tau'$.

Subcase $\texttt{Dyn} \sqsubseteq \sigma \to \tau$ and $\texttt{Dyn} \sqsubseteq \sigma' \to \tau'$. Clearly, $\texttt{Dyn} \sim \texttt{Dyn}$.

Subcase $s \to u \sqsubseteq \sigma \to \tau$ and $\texttt{Dyn} \sqsubseteq \sigma' \to \tau'$. We have $\texttt{Dyn} \sim s \to u$.

$s \to u \sqsubseteq \sigma \to \tau$ and $s' \to u' \sqsubseteq \sigma' \to \tau'$.

We have that $s \sqsubseteq \sigma$, $u \sqsubseteq \tau$, $s' \sqsubseteq \sigma'$ and $u' \sqsubseteq \tau'$. We also have that $s \sim u$ and $\sigma \sim \tau$ so by induction, we get that $s' \sim \sigma'$ and $u' \sim \sigma'$ so we are done.

Case $(\sigma \wedge \tau) \sim (\sigma' \wedge \tau')$. We proceed in the same way as the $\rightarrow$ case. $\square$

## B.3 Proof of the order isomorphism

**Theorem B.3.1** (Weakening)**.** $\forall E, \Gamma, T, T' :$ *if* $x_l \notin FV(E)$*, then* $\Gamma \vdash E : T$ *iff* $\Gamma, x : T' \vdash E : T$*.*

**Theorem 4.5.1.** $\forall E, \Gamma :$ *if* $FV(E) \subseteq Dom(\Gamma)$ *then* $(Mig_\Gamma(E), \sqsubseteq)$ *and* $(Sol(Gen(E, \Gamma, 2)), \leq)$ *are order-isomorphic.*

*Proof.* If $\varphi$ is a function from type variables to types, then we define the function $G_\varphi$ from terms to terms:

$$\begin{aligned}
G_\varphi(\texttt{true}) &= \texttt{true} \\
G_\varphi(\texttt{false}) &= \texttt{false} \\
G_\varphi(n) &= n \\
G_\varphi(x_l) &= x_l \\
G_\varphi(\lambda x : T.F) &= \lambda x : (\bigwedge_0^n \varphi(x_i^0)).G_\varphi(F) \\
G_\varphi(E_1\ E_2) &= G_\varphi(E_1)\ G_\varphi(E_2)
\end{aligned}$$

Let $E, \Gamma$ be given; they remain fixed in the remainder of the proof. Now we define the following function $\alpha_E$ with the help of $G_\varphi$:

$$\begin{aligned} \alpha_E &: Sol(Gen(E, \Gamma, 2)) \to Mig_\Gamma(E) \\ \alpha_E(\varphi) &= G_\varphi(E) \end{aligned}$$

Notice that $\Gamma$ plays no role in the definitions of $G_\varphi$ and $\alpha_E$. We will show that $\alpha_E$ is a well-defined order-isomorphism. We will do this in four steps: we will show that $\alpha_E$ is well defined, injective, and surjective, and that it preserves order.

### B.3.1 Well defined.

We will show that if $\varphi \in Sol(Gen(E, \Gamma, 2))$, then $\alpha_E(\varphi) \in Mig_\Gamma(E)$. Suppose $\varphi \in Sol(Gen(E, \Gamma, 2))$. We must show

$$E \sqsubseteq \alpha_E(\varphi) \text{ and } \exists T', \Gamma' : \Gamma' \vdash \alpha_E(\varphi) : T'.$$

In order to show $E \sqsubseteq \alpha_E(\varphi)$, notice that $E$ and $\alpha_E(\varphi)$ differ only in the type annotations of bound variables. If we have no bound variables in $E$, then $E = \alpha_E(\varphi)$. Otherwise, notice that for every occurrence of $\lambda x : \tau.F$ in $E$, we have that $\varphi \models \tau \sqsubseteq (\bigwedge_0^n x_i^0)$ and $G_\varphi(\lambda x : \tau.F) = \lambda x : (\bigwedge_0^n \varphi(x_i^0)).G_\varphi(F)$. So, we can show by induction on $E$ that $E \sqsubseteq \alpha_E(\varphi)$.

Define $Extend(\Gamma, E)$ to be $\Gamma$ extended with $(x : \tau)$ for each occurrence in $E$ of the form $\lambda x : \tau.F$. In order to show $\exists T', \Gamma' : \Gamma' \vdash \alpha_E(\varphi) : T'$, we have from Theorem B.3.1 that it is sufficient to prove the stronger statement:

$$\begin{gathered} \text{Suppose } \Gamma \vdash E : i, t \mid C. \text{ Then } \forall E' \text{ subterm of } E \text{ where } \Gamma' \vdash E' : \\ i', t' \mid C' \text{ is the subderivation } : Extend(\Gamma, G_\varphi(E)) \vdash G_\varphi(E') : \varphi([\![E'^j]\!]). \end{gathered}$$

We will proceed by induction on $E'$.

Case: $E' = \texttt{true}$. Notice that $\varphi \models [\![E']\!]^0 = \texttt{Bool}$ and use *T-True*.

Case: $E' = \texttt{false}$. Notice that $\varphi \models [\![E']\!]^0 = \texttt{Bool}$ and use *T-False*.

Case: $E' = n$. Notice that $\varphi \models [\![E']\!]^0 = \texttt{Int}$ and use *T-Num*.

Case: $E' = x_0$, where $x_0$ is free in $E$. Notice that $\varphi \models [\![E']\!]^0 = \Gamma(x)$ and $\mathit{Extend}(\Gamma, G_\varphi(E))(x) = \Gamma(x)$ and use *T-Var*.

Case: $E' = x_l$. Notice that $\varphi \models [\![x]\!]^0 = x^0$ and $\mathit{Extend}(\Gamma, G_\varphi(E))(x) = \varphi(x^0)$ and use *T-Var*.

Case: $E' = \lambda x : \tau^0.F$ where $x$ does not occur in $E'$.

Then for some $\Gamma'$, we have:

$$\frac{\Gamma \vdash E : i, t \;\; j = Tag(t) \;\; \textit{where } x \textit{ does not occur in } E \;\mid\; C}{\Gamma \vdash \lambda x : \tau^0.E : i, 1 \to t \;\mid\; C \wedge ((([\![\lambda x : \tau^0.E]\!]^j = x^0 \to [\![E]\!]^j \vee \\ [\![\lambda x : \tau^0.E]\!]^0 = x^0 \to [\![E]\!]^0) \wedge \tau^0 \sqsubseteq x^0)} \textit{ (S-Abs-2)}$$

Notice that either $\varphi \models [\![E']\!]^j = x^0 \to [\![F]\!]^j$ or $\varphi \models [\![E']\!]^0 = x^0 \to [\![F]\!]^0$ From the induction hypothesis we have $\mathit{Extend}(\Gamma, G_\varphi(E)) \vdash G_\varphi(F) : \varphi([\![F]\!]^j)$. Now we use *T-Abs-2*.

Case: $E' = \lambda x : \tau.F$ where there are $n$ occurrences of $x$ in $E'$.

We have two possibilities for the subderivation of $E'$. Suppose $i = 2$. Then we have:

$$\frac{\begin{array}{c}\Gamma \vdash E : 2, t \;\; \textit{where } x \textit{ occurs in } E \textit{ } n \textit{ times and } n \geq 1 \mid C \qquad j = Tag(t) \\ l = Tag(n \to t)\end{array}}{\begin{array}{c}\Gamma \vdash (\lambda x : \tau.E) : 2, n \to t \mid C \wedge ((([\![\lambda x : \tau.E]\!]^l = (\bigwedge^n x_i^0) \to [\![E]\!]^j \vee \\ [\![\lambda x : \tau.E]\!]^l = (\bigwedge^n x_i^0) \to [\![E]\!]^0) \wedge \\ \tau \sqsubseteq (\bigwedge^n x_i^0) \vee \\ (([\![\lambda x : \tau.E]\!]^j = x^0 \to [\![E]\!]^j \vee \\ [\![\lambda x : \tau.E]\!]^0 = x^0 \to [\![E]\!]^0) \wedge \tau \sqsubseteq x^0) \wedge x^0 = x_1^0 \wedge ... \wedge x^0 = x_n^0)\end{array}} \textit{(S-Abs-1)}$$

So we have four possibilities. Either $\varphi \models [\![\lambda x : \tau.F]\!]^l = \bigwedge^n x_i^0 \to [\![F]\!]^j$ for $j \in \{0, 2\}$ and $\varphi \models \tau \sqsubseteq \bigwedge^n x_i^0$, In this case, $Extend(\Gamma, G_\varphi(E)), (x : \tau) = Extend(\Gamma, G_\varphi(E))$. From the induction hypothesis we have $Extend(\Gamma, G_\varphi(F)) \vdash G_\varphi(F) : \varphi([\![F]\!]^j)$. Now we use *T-Abs-1*.

Otherwise, We have $\varphi \models [\![\lambda x : \tau.E]\!]^j = x^0 \to [\![F]\!]^j$ for $j \in \{0, 2\}$ and $\varphi \models \tau \sqsubseteq x^0$ and $\varphi \models x^0 = x_1^0 \wedge ... \wedge x^0 = x_n^0$.
$Extend(\Gamma, G_\varphi(E)), (x : \tau)) = Extend(\Gamma, G_\varphi(E))$. So, from the induction hypothesis we have $Extend(\Gamma, G_\varphi(E)) \vdash G_\varphi(E) : \varphi([\![E]\!]^j)$. Now we use *T-Abs-0*.

Otherwise, suppose $i = 0$. Then we have:

$$\frac{\Gamma \vdash E : 0, t \;\; \textit{where } x \textit{ occurs in } E \textit{ } n \textit{ times and } n \geq 1 \mid C}{\Gamma \vdash (\lambda x : \tau.E) : 0, 1 | C \wedge ([\![\lambda x : \tau.E]\!]^0 = x^0 \to [\![E]\!]^0 \wedge \tau \sqsubseteq x^0 \wedge x^0 = x_1^0 \wedge x^0 = x_n^0)} \textit{(S-Abs-0)}$$

We have $\varphi \models [\![\lambda x : \tau.F]\!]^0 = x^0 \to [\![E]\!]^0$ and $\varphi \models \tau \sqsubseteq x^0$ and $\varphi \models x^0 = x_1^0 \wedge ... \wedge x^0 = x_n^0$.
$Extend(\Gamma, G_\varphi(E)), (x : \tau \wedge ... \wedge \tau)) = Extend(\Gamma, G_\varphi(E))$. So, from the induction hypothesis we have $Extend(\Gamma, G_\varphi(E)) \vdash G_\varphi(E) : \varphi([\![E]\!]^0)$. Now we use *T-Abs-0*.

Case: $E' = E_1\ E_2$. Again, we have two possibilities.

$$\frac{\begin{array}{c}\Gamma \vdash E_1 : 2, t_1 \mid C_1 \qquad \Gamma \vdash E_2 : 0, t_2 \mid C_2 \\ j = Tag(Cod(t_1)) \qquad n = Dom(t_1)\ \ l = Tag(t_1)\end{array}}{\begin{array}{c}\Gamma \vdash E_1\ E_2 : 2, Cod(t_1) \mid C_1 \wedge C_2 \wedge ((([\![E_1]\!]^l \rhd \langle E_2\rangle_1^0 \wedge ... \wedge \langle E_2\rangle_n^0 \rightarrow [\![E_1\ E_2]\!]^j \\ \vee [\![E_1]\!]^l \rhd \langle E_2\rangle_1^0 \wedge ... \wedge \langle E_2\rangle_n^0 \rightarrow [\![E_1\ E_2]\!]^0) \\ \wedge \forall i \in \{1, ..., n\} : \langle E_2\rangle_i^0 \sim [\![E_2]\!]^0) \vee \\ (([\![E_1]\!]^j \rhd \langle E_2\rangle^0 \rightarrow [\![E_1\ E_2]\!]^j \vee ([\![E_1]\!]^0 \rhd \langle E_2\rangle^0 \rightarrow [\![E_1\ E_2]\!]^0) \wedge \\ \langle E_2\rangle^0 \sim [\![E_2]\!]^0))\end{array}}\ \textit{(S-App-1)}$$

From the induction hypothesis we have

$Extend(\Gamma, G_\varphi(E)) \vdash G_\varphi(E_1) : \varphi([\![E_1]\!]^j)$ and

$Extend(\Gamma, G_\varphi(E)) \vdash G_\varphi(E_2) : \varphi([\![E_2]\!]^0)$.

Then $\varphi$ must be a solution to one of the following sets of constraints:

$[\![E_1]\!]^l \rhd \langle E_2\rangle_1^0 \wedge ... \wedge \langle E_2\rangle_n^0 \rightarrow [\![E_1\ E_2]\!]^j \wedge \bigwedge_{i \leq n} \langle E_2\rangle_i^0 \sim [\![E_2]\!]^0$ for $j \in \{0, 2\}$

$[\![E_1]\!]^j \rhd \langle E_2\rangle^0 \rightarrow [\![E_1\ E_2]\!]^j \wedge \langle E_2\rangle^0 \sim [\![E_2]\!]^0$ for $j \in \{0, 2\}$

In both cases, we use *T-App*.

Otherwise, for $i = 0$, we have:

$$\frac{\Gamma' \vdash E_1 : 0, t_1 \mid C_1 \qquad \Gamma' \vdash E_2 : 0, t_2 \mid C_2}{\begin{array}{c}\Gamma' \vdash E_1\ E_2 : 0, 1 \mid C_1 \wedge C_2 \wedge [\![E_1]\!]^0 \rhd \langle E_2\rangle^0 \rightarrow [\![E_1\ E_2]\!]^0 \wedge \langle E_2\rangle^0 \sim [\![E_2]\!]^0 \wedge \\ \langle E_2\rangle^0 \sim [\![E_2]\!]^0\end{array}}\ \textit{(S-App-0)}$$

Then $\varphi \vDash [\![E_1]\!]^0 \rhd \langle E_2\rangle^0 \rightarrow [\![E_1\ E_2]\!]^0 \wedge \langle E_2\rangle^0 \sim [\![E_2]\!]^0$ and we can use *T-App*. □

### B.3.2 Injective.

We will show that $\alpha_E$ is injective, that is, we will show that

$$\text{if } \alpha_E(\varphi) = \alpha_E(\varphi'), \text{ then } \varphi = \varphi'.$$

Suppose $\alpha_E(\varphi) = \alpha_E(\varphi')$. From the definition of $\alpha_E$ we see that for every occurrence in $E$ of $\lambda x : T_1.F$, $\varphi(x^0) = \varphi'(x^0)$. We will show that for every occurrence of a subterm $E'$ in $E$ we have $\varphi([\![E']\!]^j) = \varphi'([\![E']\!]^j)$, and for every occurrence of a subterm $(E_1\ E_2)$ in $E$, we have $\varphi(\langle E_2 \rangle^j) = \varphi'(\langle E_2 \rangle^j)$ or $\varphi(t_1^0) \wedge ... \wedge \varphi(t_n^0) = \varphi'(t_1^0) \wedge ... \wedge \varphi'(t_n^0)$. We proceed by induction on $E'$.

Case: $E' = \texttt{true}$. From $\varphi \models [\![E']\!]^0 = \texttt{Bool}$ and $\varphi' \models [\![E']\!]^0 = \texttt{Bool}$, we have $\varphi([\![E']\!]^0) = \texttt{Bool} = \varphi'([\![E']\!]^0)$.

Case: $E' = \texttt{false}$. From $\varphi \models [\![E']\!]^0 = \texttt{Bool}$ and $\varphi' \models [\![E']\!]^0 = \texttt{Bool}$, we have $\varphi([\![E']\!]^0) = \texttt{Bool} = \varphi'([\![E']\!]^0)$.

Case: $E' = n$. From $\varphi \models [\![E']\!]^0 = \texttt{Int}$ and $\varphi' \models [\![E']\!]^0 = \texttt{Int}$, we have $\varphi([\![E']\!]^0) = \texttt{Int} = \varphi'([\![E']\!]^0)$.

Case: $E' = x$, where $x$ is free in $E$ From $\varphi \models [\![E']\!]^0 = \Gamma(x)$ and $\varphi' \models [\![E']\!]^0 = \Gamma(x)$, we have $\varphi([\![E']\!]^j) = \Gamma(x) = \varphi'([\![E']\!]^j)$.

Case: $E' = x$, where $x$ is bound in $E$. $\varphi \models [\![E']\!]^0 = x^0$ and $\varphi' \models [\![E']\!]^0 = x^0$, we have $\varphi([\![E']\!]^j) = \varphi(x) = \varphi'(x) = \varphi'([\![E']\!]^j)$.

Case: $E' = \lambda x : \tau^0.F$ where $x$ does not occur in $F$ From the induction hypothesis, we have $\varphi([\![F]\!]^j) = \varphi'([\![F]\!]^j)$. From $\varphi \models [\![E']\!]^i = x^0 \to [\![F]\!]^i$ and $\varphi' \models [\![E']\!]^i = x^0 \to [\![F]\!]^i$, we have $\varphi([\![E']\!]^i) = \varphi(x^0) \to \varphi([\![F]\!]^i) = \varphi'(x^0) \to \varphi'([\![F]\!]^i) = \varphi'([\![E']\!]^i)$.

Case: $E' = \lambda x : \tau^0.F$ where $x$ occurs in $F$ $n$ times.

From the induction hypothesis, we have that $\varphi([\![F]\!]^j) = \varphi'([\![F]\!]^j)$. Notice that $\varphi$ has to be a solution to a constraint of the form $[\![E']\!]^l = (\bigwedge^n x_i^0) \to [\![F]\!]^j$ and $\varphi'$ has to be a solution of the form: $[\![E']\!]^l = (\bigwedge^n x_i^0) \to [\![F']\!]^j$ Then $\varphi([\![E']\!]^l) = \varphi(x^k) \to \varphi([\![F]\!]^j) = \varphi'(x^k) \to \varphi'([\![F]\!]^j) = \varphi'([\![E']\!]^l)$.

Case: $E' = \lambda x : \tau_1 \wedge ... \wedge \tau_n.F$ where $x$ occurs in $F$ $n$ times.

From the induction hypothesis, we have $\varphi([\![F]\!]) = \varphi'([\![F]\!])$.

From $\varphi \models [\![E']\!]^j = x_1^0 \wedge ... \wedge x_n^0 \to [\![F]\!]^i$ and $\varphi' \models [\![E']\!]^j = x_1^0 \wedge ... \wedge x_n^0 \to [\![F]\!]^i$. So we have: $\varphi([\![E']\!]^j) = \varphi(x_1^0) \wedge ... \wedge \varphi(x_n^0) \to \varphi([\![F]\!]^i) = \varphi'(x_1^0) \wedge ... \wedge \varphi'(x_n^0) \to \varphi'([\![F]\!]^i) = \varphi'([\![E']\!]^j)$.

Case: $E' = E_1\ E_2$. From the induction hypothesis, we have $\varphi([\![E_1]\!]^i) = \varphi'([\![E_1]\!]^i)$ and $\varphi([\![E_2]\!]^0) = \varphi'([\![E_2]\!])^0$. From $\varphi([\![E_1]\!]) = \varphi'([\![E_1]\!])$ and $\varphi \models [\![E_1]\!]^i \rhd \langle E_2 \rangle_1^0 \wedge ... \wedge \langle E_2 \rangle_n^0 \to [\![E_1\ E_2]\!]^j$ and $\varphi' \models [\![E_1]\!]^i \rhd \langle E_2 \rangle_1^0 \wedge ... \wedge \langle E_2 \rangle_n^0 \to [\![E_1\ E_2]\!]^{j'}$. So we have $\varphi([\![E']\!]^j) = \varphi'([\![E']\!]^j)$. Another case is $\langle E_2 \rangle^k \in Dom(\varphi)$ and we can proceed in a similar way.

### B.3.3 Surjective.

We will show that $\alpha_E$ is surjective, that is, we will show that

$$\text{if } E_0 \in Mig_\Gamma(E), \text{ then } \exists \varphi \in Sol(Gen(E, \Gamma, 2)) : E_0 = \alpha_E(\varphi).$$

From $E_0 \in Mig_\Gamma(E)$ we have $E \sqsubseteq E_0$ and $\tau_0$ such that $\Gamma \vdash E_0 : \tau_0$. From $\Gamma \vdash E_0 : \tau_0$ and Theorem B.3.1, we have that $Extend(\Gamma, E_0) \vdash E_0 : \tau_0$.

We will show how to derive $\varphi$ for each subterm $E'$ and show that the result is well-typed.

Case: $E' = \texttt{true}$. From *(T-True)* we have that $\varphi([\![E']\!]^0) = \texttt{Bool}$ so $\varphi \models [\![E']\!]^0 = \texttt{Bool}$.

Case: $E' = \texttt{false}$. From *(T-False)* we have that $\varphi([\![E']\!]^0) = \texttt{Bool}$ so $\varphi \models [\![E']\!]^0 = \texttt{Bool}$ .

Case: $E' = n$. From *(T-Num)* we have that $\varphi([\![E']\!]^0) = \texttt{Int}$ so $\varphi \models [\![E']\!]^0 = \texttt{Int}$,

Case: $E' = x_0$, where $x$ is free in $E$. From *(T-Var)* we have that $\varphi([\![E']\!]) = \varphi(x_0^0) = \Gamma(x)$ so $\varphi \models [\![E']\!]^0 = \Gamma(x)$.

Case: $E' = x_l$ where $x$ is bound. From *(T-Var)* we have that $\varphi([\![E']\!]) = \varphi(x_l^0)$ so $\varphi \models [\![E']\!]^0 = x_l$.

Case: $E' = \lambda x : \tau^0.F$ where $x \notin E$. The derivation $D$ contains this use of *T-Abs-2*:

$$\frac{\Gamma \vdash F : \tau \;\; \textit{where } x_0 \textit{ does not occur in } F}{\Gamma \vdash (\lambda x : \tau^0.F) : \tau^0 \to \tau} \;\textit{(T-Abs-2)}$$

We have $\varphi(x_l^0) = \tau^0$ and $\varphi([\![F]\!]^i) = \tau$ and $\varphi([\![\lambda x : \tau^0.F]\!]) = \tau^0 \to \tau$ where $\tau$ and $\tau^0 \to \tau$ are Rank-$i$ types where $i = 0$ or $i = 2$. So, $\varphi \models [\![\lambda x : \tau^0.F]\!]^i = x^0 \to [\![F]\!]^i$. Additionally, we have $E_0 \in Mig_\Gamma(E)$ so if the type annotation of $x$ in $E$ is $S$, then we have $S \sqsubseteq \varphi(x^0)$.

Case: $E' = \lambda x : \sigma^0 \wedge ... \wedge \sigma_n.F$ where $x_i$ occurs $n$ times in $E$. The derivation $D$ contains this use of *T-Abs-1*:

$$\frac{\Gamma, (x : \sigma_0^0 \wedge ... \wedge \sigma_n^0) \vdash E : \tau \;\; \textit{where } \forall i \in \{1, ..., n\},\ x_i \textit{ occurs in } E}{\Gamma \vdash (\lambda x : \sigma_0^0 \wedge ... \wedge \sigma_n^0.E) : \sigma_0^0 \wedge ... \wedge \sigma_n^0 \to \tau} \;\textit{(T-Abs-1)}$$

So, $\bigwedge_{i \leq n} \varphi([\![x_i^0]\!]) = \sigma_i^0$ and $\varphi([\![F]\!]^i) = \tau$ and $\varphi([\![\lambda x : \sigma_0^0 \wedge ... \wedge \sigma_n.F]\!]^l) = \sigma_n^0 \wedge ... \wedge \sigma^n \to \tau$. So, $\varphi \models [\![\lambda x : \sigma_0^0 \wedge ... \wedge \sigma_n^0.F]\!]^l = (\bigwedge^n x_i^0) \to [\![F]\!]^i$ where $\tau$ is a Rank-$i$ type for $i = 0$ or $i = 2$. Finally $\sigma_0^0 \wedge ...\sigma_n^0 \to \tau$ is Rank-$l$ for $l = 0$ or $l = 1$. Additionally, we have $E_0 \in Mig_\Gamma(E)$ so if the type annotation of $x$ in $E$ is $S$, then we have $S \sqsubseteq \varphi(x)$, so $\varphi \models S \sqsubseteq (\bigwedge^n x_i^0)$ where $\sigma^0 \wedge ... \wedge \sigma_n$ is a Rank-$k$ type, $\tau$ is a Rank-$i$ type and $\sigma_0^0 \wedge ... \wedge \sigma_n^0 \to \tau$ is a Rank-$l$ type.

Case: $E' = \lambda x : \sigma^0.F$ where $x$ occurs $n$ times in $E$. The derivation $D$ contains this use of *T-Abs-0*:

$$\frac{\begin{array}{c}\Gamma, (x : \sigma_0^0 \wedge \ldots \wedge \sigma_n^0) \vdash_S E : \tau \\ \textit{where } \forall i \in \{1, \ldots, n\}, x_i \textit{ occurs in } E \\ \textit{and } \sigma_0^0 = \ldots = \sigma_n^0 = \sigma^0\end{array}}{\Gamma \vdash_S (\lambda x : \sigma^0.E) : \sigma^0 \to \tau} \;\textit{(T-Abs-0)}$$

So, $\bigwedge_{i \leq n} \varphi(x_i) = \sigma^i$ and $\varphi([\![F]\!]^i) = \tau$ and $\varphi([\![\lambda x : \tau^0.F]\!]^i) = \sigma^0 \to \tau$. So, $\varphi \models [\![\lambda x : \sigma^0.F]\!]^i = x^0 \to [\![F]\!]^i$ where $i = 0$ or $i = 2$. Additionally, we have $E_0 \in Mig_\Gamma(E)$ so if the type annotation of $x$ in $E$ is $S$, then we have $S \sqsubseteq \sigma^0 = \varphi(x^0)$, so $\varphi \models S \sqsubseteq x^0$.

Case: $E' = E_1\ E_2$. The derivation $D$ contains this use of *T-App*:

$$\frac{\begin{array}{cc}\Gamma \vdash E_1 : \sigma_1 & \Gamma \vdash E_2 : \sigma^0 \\ \sigma_1 \rhd ((\bigwedge_{i \in I} \tau_i) \to \sigma) & \forall i \leq I : \sigma^0 \sim \tau_i\end{array}}{\Gamma \vdash E_1\ E_2 : \sigma}\ \textit{(T-App)}$$

Then we have $\varphi([\![E_1]\!]^j) = \sigma_1$ where $\sigma_1$ is Rank-$j$. And $\varphi([\![E_2]\!]^0) = \sigma^0$ and $\varphi(\langle E_2 \rangle^0) = (\bigwedge_{i \in I} \tau_i)$. Then let $\bigwedge_{i \leq n} \varphi(t_i^0) = \tau_i$.
Finally, let $\varphi([\![E_1\ E_2]\!]^k) = \sigma$. So $\varphi \vDash [\![E_1\ E_2]\!]^k : \sigma$ where $\sigma$ is a Rank-$k$ type where $k = 0$ or $k = 1$.

### B.3.4 Preserves order.

We will show that $\alpha_E$ preserves order, that is, we will show that

$$\text{if } \varphi \leq \varphi', \text{ then } \alpha_E(\varphi) \sqsubseteq \alpha_E(\varphi').$$

We will prove the following stronger statement:

$$\text{if } \varphi \leq \varphi', \text{ then } \forall E' : G_\varphi(E') \sqsubseteq G_{\varphi'}(E').$$

Suppose that $\varphi \leq \varphi'$. We proceed by induction on $E'$.

Case: $E' = \texttt{true}$. We have $G_\varphi(E') = \texttt{true} = G_{\varphi'}(E')$.

Case: $E' = \texttt{false}$. We have $G_\varphi(E') = \texttt{false} = G_{\varphi'}(E')$.

Case: $E' = n$. We have $G_\varphi(E') = n = G_{\varphi'}(E')$.

Case: $E' = x_l$. We have $G_\varphi(E') = x_l = G_{\varphi'}(E')$.

Case: $E' = \lambda x : T.F$. We have two cases. Either $x \in Dom(E')$. In this case, from induction hypothesis, we have $G_\varphi(F) \sqsubseteq G_{\varphi'}(F)$. From $\varphi \leq \varphi'$ we have $\varphi(x_l^0) \sqsubseteq \varphi'(x_l^0)$. From the definition of $G_\varphi$ and *(P-Abs)* we have $G_\varphi(\lambda x : T.F) = \lambda x : \varphi(x_l^0).G_\varphi(F) \sqsubseteq \lambda x : \varphi'(x_l^0).G_{\varphi'}(F) = G_{\varphi'}(\lambda x : T.F)$. Otherwise, $x_1^0 \in Dom(\varphi) \wedge ... \wedge x_n^0 \in Dom(\varphi)$. We again have that $G_\varphi(F) \sqsubseteq G_{\varphi'}(F)$. From $\varphi \leq \varphi'$ we have $\varphi(x_i^0) \sqsubseteq \varphi'(x_i^0)$ for all $i \leq$ n. From the definition of $G_\varphi$ and *(P-Abs)* we have $G_\varphi(\lambda x : T.F) = \lambda x : \bigwedge_0^n \varphi(x_i^0).G_\varphi(F) \sqsubseteq \lambda x : \bigwedge_n^0 \varphi'(x_i^0).G_{\varphi'}(F) = G_{\varphi'}(\lambda x : T.F)$.

Case: $E' = E_1\ E_2$. From induction hypothesis, we have $G_\varphi(E_1) \sqsubseteq G_{\varphi'}(E_1)$ and $G_\varphi(E_2) \sqsubseteq G_{\varphi'}(E_2)$. From the definition of $G_\varphi$ and *(P-App)* we have $G_\varphi(E_1\ E_2) = G_\varphi(E_1)\ G_\varphi(E_2) \sqsubseteq G_{\varphi'}(E_1)\ G_{\varphi'}(E_2) = G_{\varphi'}(E_1\ E_2)$.

## B.4 Constraints

### B.4.1 Full grammars

$$
\begin{aligned}
PEC \ ::= \ & PEC_1 \wedge PEC_2 \ \mid \ PEC_1 \vee PEC_2 \ \mid \ \tau \sqsubseteq (v_1^0 \wedge \ldots \wedge v_n^0) \ \mid \ v_1^0 \sim v_2^0 \ \mid \\
& v^0 = \tau^0 \ \mid \ v_1^0 = v_2^0 \ \mid \ v^0 = v_1^0 \to v_2^0 \ \mid \\
& v_1^2 = (v_1^0 \wedge \ldots \wedge v_n^0) \to v_2^2 \ \mid \ v^2 = (v_1^0 \wedge v_1^2 \ldots \wedge v_n^0) \to v^0
\end{aligned}
$$

$$
\begin{aligned}
EC \quad ::= \quad & EC_1 \wedge EC_2 \;\mid\; EC_1 \vee EC_2 \;\mid\; v_1^0 \sim v_2^0 \;\mid \\
& v^0 = \tau^0 \;\mid\; v_1^0 = v_2^0 \;\mid\; v^0 = v_1^0 \to v_2^0 \;\mid \\
& v_1^2 = (v_1^0 \wedge \ldots \wedge v_n^0) \to v_2^2 \;\mid\; v^2 = (v_1^0 \wedge v_1^2 \ldots \wedge v_n^0) \to v^0
\end{aligned}
$$

$$
DNF(EC) \textit{ is the } DNF \textit{ of } EC
$$

$$
C_\sim := C_{1\sim} \wedge C_{2\sim} | C_{1\sim} \vee C_{2\sim} | v^0 \sim v^{0\prime} | v^0 \sim \tau | \tau_1 \sim \tau_2
$$

$$
C_- := C_{1-} \wedge C_{2-} | C_{1-} \vee C_{2-} | v^0 \sim \tau
$$

### B.4.2 Constraint solver

We refer to the set $PEC$ as a constraint system $C$ involving Precision, Equality and Consistency.

We will define a series of transformations which simplify our constraints but maintain the set of solutions. $PEC$, $EC$ and $C_\sim$ are similar to those in [MP19].

#### B.4.2.1 Precision constraints.

We define a simplification procedure *SimPrec* that transforms every Precision constraint into zero, one, or more Equality constraints:

$$
SimPrec \quad : \quad PEC \to EC
$$

We define *SimPrec* to leave the set of type variables unchanged, and to proceed by repeating the following transformation until it no longer has an effect:

| From | To |
|---|---|
| $PEC_1 \vee PEC_2$ | $SimPrec(PEC_1) \vee SimPrec(PEC_2)$ |
| $PEC_1 \wedge PEC_2$ | $SimPrec(PEC_1) \wedge SimPrec(PEC_2)$ |
| $\texttt{Dyn} \sqsubseteq v^0$ | *True* |
| $\texttt{Bool} \sqsubseteq v^0$ | $v^0 = \texttt{Bool}$ |
| $\texttt{Int} \sqsubseteq v^0$ | $v^0 = \texttt{Int}$ |
| $w_1^0 \to w_2^0 \sqsubseteq v^0$ | $v^0 = v_1^0 \to v_2^0$ |
| | $w_1^0 \sqsubseteq v_1^0$ |
| | $w_2^0 \sqsubseteq v_2^0$ |
| | where $v_1^0, v_2^0$ are fresh type variables |
| $\texttt{Dyn} \sqsubseteq v_1^0 \wedge ... \wedge v_n^0 \mid n > 1$ | *True* |
| $\texttt{Int} \sqsubseteq v_1^0 \wedge ... \wedge v_n^0 \mid n > 1$ | *False* |
| $\texttt{Bool} \sqsubseteq v_1^0 \wedge ... \wedge v_n^0 \mid n > 1$ | *False* |
| $w_1^0 \to w_2^0 \sqsubseteq v_1^0 \wedge ... \wedge v_n^0$ | *False* |
| $w_1^0 \wedge ... \wedge w_n^0 \sqsubseteq v_1^0 \wedge ... \wedge v_n^0$ | $w_1^0 \sqsubseteq v_1^0, ..., w_n^0 \sqsubseteq v_n^0$ |
| $w_1^0 \wedge ... \wedge w_n^0 \sqsubseteq v_1^0 \wedge ... \wedge v_n^0 \mid m \neq n$ | *False* |
| $c$ | $c$ |

**Theorem B.4.1.** $\forall A \in PEC : Sol(A) = Sol(SimPrec(A))$.

*Proof.* Straightforward. □

**Theorem B.4.2.** $\forall A \in EC : Sol(A) = Sol(TR(A))$.

*Proof.* Since tags represent the lowest possible rank of a variable, a variable may not have the two different ranks simultaneously, so removing a clause where a variable has conflicting ranks maintains solvability. Furthermore, in our DNF, having $False$ in a clause implies that it evaluates to $False$, so we can remove it. □

#### B.4.2.2 Equality constraints.

We define *Subst* as a disjunction of mappings of substitutions that have domain *TypeVar* and range *TypeExp*. For a substitution $\sigma$, we define $Dom(\sigma)$ to be the set of type variables $v$ such that $\sigma(v) \neq v$. We use $\sigma \cup \sigma'$ to denote the union of two substitutions $\sigma, \sigma'$ that have disjoint domains. In particular, we have the following grammar:

$Subst := \sigma | Subst \vee Subst$

We define a function *Unify* that solves the Equality constraints and ignores the consistency constraints for each clause.

$$Unify \quad : \quad EC \rightarrow subst$$

We define $Unify(DNF(EC))$ to produce the most general unifier (MGU) of each clause of conjunctions in $DNF(EC)$. if no solution exists, we remove that clause from our constraints.

In particular, we have that $DNF(EC)$ has the form $(x_1 \wedge ... \wedge x_n) \vee ... \vee (y_1 \wedge ... \wedge y_m)$ with $k$ clauses, then we output $\sigma_1 \vee ... \vee \sigma_k$ where $\sigma_i$ is the $MGU$ corresponding to the $i^{th}$ clause for $i \leq k$.

**Theorem B.4.3.**

$$\forall A \in EC : Sol(A) = \begin{cases} \{\ (\sigma \circ \sigma') \cup \sigma' \ \mid \ \sigma' \in Sol(SimEq(A, \sigma))\ \} & \text{if } \sigma \neq fail \\ \emptyset & \text{if } \sigma = fail \end{cases}$$

*where* $\sigma = Unify(A)$.

*Proof.* The proof is the same as that in [MP19]. □

We define a function *SimEq* that uses *Subst* transform away all Equality constraints for each corresponding clause of conjunctions. If a solution is $False$, we write $False$.

$$SimEq \quad : \quad (EC' \times Subst) \rightarrow C_{\sim}$$

We define $SimEq(A', S)$ as follows. For each clause in $A'$ and $S$, Let $A_i$ be the $i^{th}$ clause in $A'$. The set of type variables is $vars(A_i) \setminus Dom(\sigma_i)$ where $\sigma_i$ is the $i^{th}$ clause in $S$. Second, the set of constraints consists of only Consistency constraints: apply the substitution to the Consistency constraints in $A_i$ and return only those transformed Consistency constraints.

**Theorem B.4.4.** $\forall A \in EC : Sol(A)$ *is finite iff* ($\sigma \neq fail$ *implies* $Sol(SimEq(A, \sigma))$ *is finite),* *where* $\sigma = Unify(A)$.

*Proof.* Immediate from Theorem B.4.3. □

#### B.4.2.3 Consistency constraints.

We define a function *SimCon* that simplifies a formula consistency constraints.

$$SimCon \quad : \quad C_{\sim} \rightarrow C_{\sim}$$

We define *SimCon* by repeatedly applying the following transformations to each clause until no transformation applies.

| From | To |
|---|---|
| $C_1 \wedge C_2$ | $SimCon(C_1) \wedge SimCon(C_2)$ |
| $C_1 \wedge C_2$ | $SimCon(C_1) \vee SimCon(C_2)$ |
| $\texttt{Bool} \sim \texttt{Bool}$ | $True$ |
| $\texttt{Int} \sim \texttt{Int}$ | $True$ |
| $\tau^0 \sim \texttt{Dyn}$ | $True$ |
| $\texttt{Dyn} \sim \tau^0$ | $True$ |
| $(\tau_1^0 \to \tau_2^0) \sim \texttt{Bool}$ | $False$ |
| $(\tau_1^0 \to \tau_2^0) \sim \texttt{Int}$ | $False$ |
| $\texttt{Bool} \sim (\tau_1^0 \to \tau_2^0)$ | $False$ |
| $\texttt{Int} \sim (\tau_1^0 \to \tau_2^0)$ | $False$ |
| $\texttt{Bool} \sim \texttt{Int}$ | $False$ |
| $\texttt{Int} \sim \texttt{Bool}$ | $False$ |
| $(\tau_1 \to \tau_2) \sim (\tau_1^{0'} \to \tau_2^{0'})$ | $(\tau_1^0 \sim \tau_1^{0'}) \wedge (\tau_2^0 \sim \tau_2^{0'})$ |
| $\tau^0 \sim v^0$ | $v^0 \sim \tau$ |

**Theorem B.4.5.**

$$\forall A \in C : Sol(A) = \begin{cases} Sol(SimCon(A)) & \textit{if } SimCon(A) \neq \textit{fail} \\ \emptyset & \textit{otherwise} \end{cases}$$

*Proof.* Straightforward. □

We finally perform the boundedness check in the same way as in the previous work. Here, we only consider the variables corresponding to the particular set we are looking at. In

particular, we apply the following predicate to all clauses $A_i$ in $A$ and expect the predicate to be true for all clauses for the space to be finite.

$$\begin{aligned} \mathit{Bounded} &: C^- \to \mathit{Boolean} \\ \mathit{Bounded}(A_i) &= \forall v, \in \mathit{vars}(A_i) : \exists T \in \mathit{MaximumType} : \mathit{BoundedVar}(v, T, A_i) \end{aligned}$$

**Theorem B.4.6.** *For $A \in C^-$: $Sol(A)$ is finite iff $Bounded(A)$.*

*Proof.* The proof is the same as [MP19]. □

Note that in our solver, once we modify precision the precision constraints in the way we described above, which is similar to the case of GTLC, we can replace matching constraints with a conjunction of the two matching possibilities. At this point we are ready to write our constraints in DNF and deal with every clause separately to determine finiteness. As we said, since a single variable should not have different tags in a valid solution, conflicting tags in one clause means that particular clause has no solution and we simply eliminate it. This is still exponential complexity as before, since in the GTLC we are only dealing with matching constraints. Here we are simply dealing with more cases.

**Theorem 4.5.2.** *We can solve the Finiteness problem in EXPTIME.*

Observe the definition of our solver above and note that the bottleneck is the step where we convert our constraint into DNF. The rest of the steps have the same complexity as the solver from the GTLC.

### B.4.3 The Singleton problem

The following theorems also hold for gradual Rank-2. The proof is the same as that for the GTLC.

**Theorem B.4.7.** $\forall T : \mathcal{S}^1(T) = \{\ \tau_u \ \mid\ \tau_u \neq T\ \wedge\ \forall T' : (T \sqsubseteq T' \sqsubseteq \tau_u)\ \textit{iff}\ ((T = T') \vee (T' = \tau_u))\ \}$.

**Theorem B.4.8.** $\forall E : \mathcal{S}(E) = \{\ E_u \ \mid\ E_u \neq E \wedge \forall E' : (E \sqsubseteq E' \sqsubseteq E_u)\ \textit{iff}\ ((E = E') \vee (E' = E_u))\ \}$.

## B.5 Calculi and Runtime Semantics

Syntax:

$$\begin{array}{rrcl} \text{(Types)} & T & ::= & \texttt{Dyn} \mid \texttt{Bool} \mid \texttt{Int} \mid T \to T \\ \text{(Terms)} & E & ::= & \texttt{true} \mid \texttt{false} \mid n \mid x \mid \lambda x : T.E \mid E\ E \\ \text{(Environments)} & \Gamma & ::= & \emptyset \mid \Gamma, x : T \end{array}$$

Typing rules:

$\Gamma \vdash_G \texttt{true} : \texttt{Bool}$ *(T-True)* $\quad \Gamma \vdash_G \texttt{false} : \texttt{Bool}$ *(T-False)* $\quad \Gamma \vdash_G n : \texttt{Int}$ *(T-Num)*

$$\frac{x : T \in \Gamma}{\Gamma \vdash_G x : T}\ (T\text{-}Var) \qquad \frac{\Gamma, x : T_1 \vdash_G E : T_2}{\Gamma \vdash_G (\lambda x : T_1.E) : T_1 \to T_2}\ (T\text{-}Abs)$$

$$\frac{\begin{array}{cc} \Gamma \vdash_G E_1 : T_1 & \Gamma \vdash_G E_2 : T_2 \\ T_1 \rhd (T_{11} \to T_{12}) & T_2 \sim T_{11} \end{array}}{\Gamma \vdash_G E_1\ E_2 : T_{12}}\ (T\text{-}App)$$

Consistency:

$T \sim \texttt{Dyn}$ *(C-Dyn1)* $\quad \texttt{Dyn} \sim T$ *(C-Dyn2)* $\quad \texttt{Bool} \sim \texttt{Bool}$ *(C-Bool)*

$\texttt{Int} \sim \texttt{Int}$ *(C-Int)*

$$\frac{T_1 \sim T_3 \quad T_2 \sim T_4}{(T_1 \to T_2) \sim (T_3 \to T_4)}\ (C\text{-}Arrow)$$

Matching:

$(T_1 \to T_2) \rhd (T_1 \to T_2)$ *(M-Arrow)* $\quad \texttt{Dyn} \rhd (\texttt{Dyn} \to \texttt{Dyn})$ *(M-Dyn)*

Precision:

$\texttt{Dyn} \sqsubseteq_G T$ *(P-Dyn)* $\quad T \sqsubseteq_G T$ *(P-Refl-T)* $\quad \dfrac{T_1 \sqsubseteq_G T_3 \quad T_2 \sqsubseteq_G T_4}{T_1 \to T_2 \sqsubseteq_G T_3 \to T_4}$ *(P-Arrow)*

$E \sqsubseteq_G E$ *(P-Refl-E)* $\quad \dfrac{T_1 \sqsubseteq_G T_2 \quad E_1 \sqsubseteq_G E_2}{\lambda x : T_1.E_1 \sqsubseteq_G \lambda x : T_2.E_2}$ *(P-Abs)*

$$\frac{E_1 \sqsubseteq_G E_3 \quad E_2 \sqsubseteq_G E_4}{(E_1\ E_2) \sqsubseteq_G (E_3\ E_4)}\ (P\text{-}App)$$

**Figure B.2:** The gradually typed $\lambda$-calculus.

### B.5.1 The Cast Calculus CC

**The Cast Calculus CC extends the STLC:**

Typing Rules:

$\Gamma \vdash_{CC} E : T.$

$$\Gamma \vdash_{CC} \texttt{true} : \texttt{Bool} \qquad \Gamma \vdash_{CC} \texttt{false} : \texttt{Bool} \qquad \Gamma \vdash_{CC} n : \texttt{Int}$$

$$\frac{x : T \in \Gamma}{\Gamma \vdash_{CC} x : T} \qquad \Gamma \vdash_{CC} blame_T \mathcal{L} : T$$

$$\frac{\Gamma, (x : T_1) \vdash_{CC} E : T_2}{\Gamma \vdash_{CC} (\lambda x : T_1 . E) : T_1 \to T_2}$$

$$\frac{\Gamma \vdash_{CC} E_1 : \sigma^0 \to \sigma \qquad \Gamma \vdash_{CC} E_2 : \sigma^0}{\Gamma \vdash_{CC} E_1\ E_2 : \sigma}$$

$$\frac{\Gamma \vdash_{CC} E : T_1}{\Gamma \vdash_{CC} E : (T_1 \Rightarrow^{\mathcal{L}} T_2) : T_2}$$

**Compilation from GTLC to CC:**

$\Gamma \vdash_{CC} E \rightsquigarrow E' : T.$

$$\Gamma \vdash_{CC} \texttt{true} \rightsquigarrow \texttt{true} : \texttt{Bool} \qquad \Gamma \vdash_{CC} \texttt{false} \rightsquigarrow \texttt{false} : \texttt{Bool} \qquad \Gamma \vdash_{CC} n \rightsquigarrow n : \texttt{Int}$$

$$\frac{x : T \in \Gamma}{\Gamma \vdash_{CC} x \rightsquigarrow x : T}$$

$$\frac{\Gamma, x : T_1 \vdash_{CC} E \rightsquigarrow E' : T_2}{\Gamma \vdash_{CC} \lambda x : T_1 . E \rightsquigarrow \lambda x : T_1 . E' : T_1 \to T_2}$$

$$\frac{\begin{array}{cc} \Gamma \vdash_{CC} E_1 \rightsquigarrow E_1' : \sigma_1 & \Gamma \vdash_{CC} E_2 \rightsquigarrow E_2' : \sigma_2 \\ \sigma_1 \rhd (T \to \sigma) & \sigma_2 \sim T \end{array}}{\Gamma \vdash_{CC} E_1\ E_2 \rightsquigarrow (E_1' : \sigma_1 \Rightarrow^{\mathcal{L}_0} (T \to \sigma))(E_2' : \sigma^0 \Rightarrow^{\mathcal{L}_1} T) : \sigma}$$

**Figure B.3:** CC

### B.5.2 Runtime Semantics

$$
\begin{array}{rrcl}
\text{(Ground Types)} & G & ::= & \texttt{Dyn} \to \texttt{Dyn} \mid \texttt{Bool} \mid \texttt{Int} \\
\text{(Terms)} & E, f & ::= & \mid \{ \mid n \mid x \mid \lambda x : T.E \mid E\ E \mid E : T \Rightarrow^{\mathcal{L}} T \mid \texttt{blame}_T\mathcal{L} \\
\text{(Values)} & v & ::= & \texttt{true} \mid \texttt{false} \mid n \mid x \mid \lambda x : T.E \mid v : T_1 \to T_2 \Rightarrow^{\mathcal{L}} T_3 \to T_4 \mid \\
v : G \Rightarrow^{\mathcal{L}} \texttt{Dyn} & & & \\
 & r & ::= & v \mid \texttt{blame}_T\mathcal{L} \\
 & F & ::= & \Box f \mid v\Box \mid \Box : T_1 \Rightarrow^{\mathcal{L}} T_2
\end{array}
$$

$$(\lambda x : T.f)v \mapsto [x := v]f$$

$$v : B \Rightarrow^{\mathcal{L}} B \mapsto v$$

$$v : \texttt{Dyn} \Rightarrow^{\mathcal{L}} \texttt{Dyn} \mapsto v$$

$$v : G \Rightarrow^{\mathcal{L}_1} \texttt{Dyn} \Rightarrow^{\mathcal{L}_2} G \mapsto v$$

$$v : G_1 \Rightarrow^{\mathcal{L}_1} \texttt{Dyn} \Rightarrow^{\mathcal{L}_2} G_2 \mapsto \texttt{blame}_{G_2}\mathcal{L}_2 \ \textit{where}\ G_1 \neq G_2$$

$$(v_1 : T_1 \to T_2 \Rightarrow^{\mathcal{L}} T_3 \to T_4)v_2 \mapsto v_1(v_2 : T_3 \Rightarrow^{\mathcal{L}} T_1) : T_2 \Rightarrow^{\mathcal{L}} T_4$$

$$v : T \Rightarrow^{\mathtt{L}} \mathtt{Dyn} \mapsto v : T \Rightarrow^{\mathtt{L}} G \Rightarrow^{\mathtt{L}} \mathtt{Dyn} \ \mathit{if}\ T \neq \mathtt{Dyn}, T \neq G, T \sim G$$

$$v : \mathtt{Dyn} \Rightarrow^{\mathcal{L}} T \mapsto v : \mathtt{Dyn} \Rightarrow^{\mathcal{L}} G \Rightarrow^{\mathcal{L}} T \ \mathit{if}\ T \neq \mathtt{Dyn}, T \neq G, T \sim G$$

$$F[f] \mapsto F[f'] \ \mathit{if}\ f \mapsto f'$$

$$F[\mathtt{blame}_{T_1}\mathcal{L}] \mapsto \mathtt{blame}_{T_2}\mathcal{L} \ \mathit{if}\ \Gamma \vdash F : T_1 \Rightarrow T_2$$

## B.6 Flexible Rank-2: Adding Associativity and Commutativity

**Definition B.2** (Type Equivalence)**.**

$$\tau \wedge \sigma \cong \sigma \wedge \tau$$

$$\tau_1 \wedge (\tau_2 \wedge \tau_3) \cong (\tau_1 \wedge \tau_2) \wedge \tau_3$$

Typing Rules

$\Gamma \vdash_f E : T^2$.

$\Gamma \vdash_f \texttt{true} : \texttt{Bool}$ *(T-True)* $\quad$ $\Gamma \vdash_f \texttt{false} : \texttt{Bool}$ *(T-False)* $\quad$ $\Gamma \vdash_f n : \texttt{Int}$ *(T-Num)*

$$\frac{x : \bigwedge_{i \in I} \tau_i^0 \in \Gamma \quad l \in I}{\Gamma \vdash_f x_l : \tau_l^0} \ (T\text{-}Var)$$

$$\frac{\begin{array}{c}\Gamma, (x : \sigma_0^0 \wedge \ldots \wedge \sigma_n^0) \vdash_f E : \tau \\ \textit{where } \forall i \in 0..n, x_i \textit{ occurs in } E \\ \textit{and } \sigma_0^0 = \ldots = \sigma_n^0 = \sigma^0\end{array}}{\Gamma \vdash_f (\lambda x : \sigma^0.E) : \sigma^0 \to \tau} \ \textit{(T-Abs-0)}$$

$$\frac{\begin{array}{c}\Gamma, (x : \sigma_0^0 \wedge \ldots \wedge \sigma_n^0) \vdash_f E : \tau \\ \textit{where } \forall i \in 0..n, x_i \textit{ occurs in } E \\ \textit{and } \sigma \cong \sigma_0^0 \wedge \ldots \wedge \sigma_n^0\end{array}}{\Gamma \vdash_f (\lambda x : \sigma.E) : \sigma \to \tau} \ \textit{(T-Abs-1-f)}$$

$$\frac{\Gamma \vdash_f E : \tau \ \ \textit{where } x_0 \textit{ does not occur in } E}{\Gamma \vdash_f (\lambda x : \tau^0.E) : \tau^0 \to \tau} \ \textit{(T-Abs-2)}$$

$$\frac{\begin{array}{cc}\Gamma \vdash_f E_1 : \sigma_1 & \Gamma \vdash_f E_2 : \sigma^0 \\ \sigma_1 \rhd ((\bigwedge_{i \in I} \tau_i^0) \to \sigma) & \forall i \leq I : \sigma^0 \sim \tau_i^0\end{array}}{\Gamma \vdash_f E_1 \ E_2 : \sigma} \ \textit{(T-App)}$$

Consistency:

$\sigma^0 \sim \texttt{Dyn}$ *(C-Dyn1)* $\quad$ $\texttt{Dyn} \sim \sigma^0$ *(C-Dyn2)* $\quad$ $\texttt{Bool} \sim \texttt{Bool}$ *(C-Bool)*

$\texttt{Int} \sim \texttt{Int}$ *(C-Int)*

$$\frac{\sigma^0 \sim \sigma'^0 \quad \tau^0 \sim \tau'^0}{(\sigma^0 \to \tau^0) \sim (\sigma'^0 \to \tau'^0)} \ \textit{(C-Arrow)}$$

Matching:

$(\sigma^1 \to \tau^2) \rhd (\sigma^1 \to \tau^2)$ *(M-Arrow)* $\quad$ $\texttt{Dyn} \rhd (\texttt{Dyn} \to \texttt{Dyn})$ *(M-Dyn)*

**Figure B.4:** The Flexible Gradual Rank-2 intersection-typed $\lambda$-calculus: Typing Rules

**Definition B.3** (Term Equivalence)**.**

$$E \cong E$$

$$\lambda x : \tau.E \cong \lambda x : \tau'.E' \iff E \cong E' \wedge \tau \cong \tau'$$

$$E_1\ E_2 \cong E_1'\ E_2' \iff E_1 \cong E_1' \wedge E_2 \cong E_2'$$

**Definition B.4** (Flexible Type Precision)**.** $\tau \lesssim \sigma$ iff $\exists \tau' : \tau \cong \tau' \wedge \tau' \sqsubseteq \sigma$

**Definition B.5** (Flexible Term Precision)**.** $E \lesssim E'$ iff $\exists E'' : E \cong E'' \wedge E'' \sqsubseteq E'$

**Definition B.6** (Flexible Migration)**.** $E'$ is a flexible $\Gamma$-migration of $E$ (written $E \leq_{\Gamma_f} E'$) iff $(E \lesssim E' \ \wedge \ \exists T' : \Gamma \vdash_f E' : T')$.

**Theorem 4.7.5.** *Let* $\Gamma \vdash E_s : T_s$ *and* $\Gamma \vdash_f E : T$ *and* $E_s \cong E$ *and* $T_s \cong T$*. Then for every* $E' \lesssim E$*, there exists some* $E_s' \cong E'$ *with* $E_s' \sqsubseteq E_s$*, where* $\forall \Gamma'$*, if* $\Gamma' \vdash E_s' : T_s'$*, then* $\Gamma' \vdash_f E' : T'$ *where* $T' \lesssim T$*,* $T_s' \sqsubseteq T_s$*,* $\Gamma' \sqsubseteq \Gamma$ *and* $T' \cong T_s'$*.*

*Proof.* Cases *T-True*, *T-False*, *T-Num*, *T-Var* are straightforward.

Case *T-Abs-0*

We have:

$$\frac{\begin{array}{c}\Gamma, (x : \sigma_0^0 \wedge \ldots \wedge \sigma_n^0) \vdash_f E : \tau \\ \textit{where } \forall i \in 0..n, x_i \textit{ occurs in } E \\ \textit{and } \sigma_0^0 = \ldots = \sigma_n^0 = \sigma^0\end{array}}{\Gamma \vdash_f (\lambda x : \sigma^0.E) : \sigma^0 \to \tau}\ \textit{(T-Abs-0)}$$

We also have:

$$\frac{\begin{array}{l}\Gamma, (x : \sigma^0_{0'} \wedge \ldots \wedge \sigma^0_{n'}) \vdash E' : \tau' \\ \textit{where } \forall i \in \{0, \ldots, n'\}, x_i \textit{ occurs in } E' \\ \textit{and } \sigma^0_{0'} = \ldots = \sigma^0_{n'} = \sigma^{0'}\end{array}}{\Gamma \vdash (\lambda x : \sigma^{0'}.E') : \sigma^{0'} \to \tau'} \textit{ (T-Abs-0)}$$

So $\Gamma, (x : \sigma^0_0 \wedge ... \wedge \sigma^0_n) \vdash_f E : \tau$ and $\Gamma, (x : \sigma^0_{0'} \wedge ... \wedge \sigma^0_{n'}) \vdash E' : \tau'$ and $E \cong E'$ and $\tau \cong \tau'$. Then for every $E_f \lesssim E$, there exists some $E_s \cong E_f$ with $E_s \sqsubseteq E'$, where if $\Gamma', (x : \sigma^0_{0''} \wedge ... \wedge \sigma^0_{n''}) \vdash E_s : T_s$, then $\Gamma', (x : \sigma^0_{0''} \wedge ... \wedge \sigma^0_{n''}) \vdash_f E_f : T'$ where $T' \lesssim \tau$ and $T_s \sqsubseteq \tau'$ and $\Gamma', (x : \sigma^0_{0''} \wedge ... \wedge \sigma^0_{n'}) \sqsubseteq \Gamma', (x : \sigma^0_{0'} \wedge ... \wedge \sigma^0_{n'})$. We also have that $T' \cong T_s$.

Then notice that by the definition of equivalence on types that $\sigma^0_0 \wedge ... \wedge \sigma^0_n = \sigma^{0'}$. So we can apply the induction hypothesis along with *T-Abs-0* to get that for every $\lambda x : \sigma^{0''}.E_f \lesssim \lambda x : \sigma^0.E$, we have $\lambda x : \sigma^{0''}.E_s \cong \lambda x : \sigma^{0''}.E_f$ with $\lambda x : \sigma^{0''}.E_s \sqsubseteq \lambda x : \sigma^0.E'$, where if $\Gamma' \vdash \lambda x : \sigma^{0''}.E_s : \sigma^{0''} \to T_s$, then $\Gamma' \vdash_f \lambda x : \sigma^{0''}.E_f : \sigma^{0''} \to T'$ where $\sigma^{0''} \to T_s \sqsubseteq \sigma^0 \to \tau'$ and $\sigma^{0''} \to T' \lesssim \sigma^0 \to \tau$ and $\sigma^{0''} \to T' \cong \sigma^{0''} \to T_s$ and $\Gamma' \sqsubseteq \Gamma$.

Case *T-Abs-1-f* we have:

$$\frac{\begin{array}{l}\Gamma, (x : \sigma^0_0 \wedge \ldots \wedge \sigma^0_n) \vdash_f E : \tau \\ \textit{where } \forall i \in 0..n, x_i \textit{ occurs in } E \\ \textit{and } \sigma \cong \sigma^0_0 \wedge \ldots \wedge \sigma^0_n\end{array}}{\Gamma \vdash_f (\lambda x : \sigma.E) : \sigma \to \tau} \textit{ (T-Abs-1-f)}$$

$$\frac{\Gamma, (x : \sigma_0^0 \wedge ... \wedge \sigma_n^0) \vdash E' : \tau' \;\; \textit{where } \forall i \in 0..n,\ x_i \textit{ occurs in } E'}{\Gamma \vdash (\lambda x : \sigma_0^0 \wedge ... \wedge \sigma_n^0 . E') : \sigma_0^0 \wedge ... \wedge \sigma_n^0 \to \tau'} \textit{ (T-Abs-1)}$$

So $\Gamma, (x : \sigma_0^0 \wedge ... \wedge \sigma_n^0) \vdash_f E : \tau$ and $\Gamma, (x : \sigma_0^0 \wedge ... \wedge \sigma_n^0) \vdash E' : \tau'$ and $E \cong E'$ and $\tau \cong \tau'$. Then for every $E_f \lesssim E$, there exists some $E_s \cong E_f$ with $E_s \sqsubseteq E'$, where if $\Gamma', (x : \sigma_{1'}^0 \wedge ... \wedge \sigma_{n'}^0) \vdash E_s : T_s$, then $\Gamma', (x : \sigma_{1'}^0 \wedge ... \wedge \sigma_{n'}^0) \vdash_f E_f : T'$ where $T' \lesssim \tau$ and $T_s \sqsubseteq \tau'$. We also have that $T' \lesssim T_s$. We also have that $\Gamma', (x : \sigma_{1'}^0 \wedge ... \wedge \sigma_{n'}^0) \sqsubseteq \Gamma, (x : \sigma_1^0 \wedge ... \wedge \sigma_n^0)$.

Then notice that we have that $\sigma \cong \sigma_0^0 \wedge ... \wedge \sigma_n^0$ and that our induction hypothesis holds for every $\Gamma', (x : \sigma_{1'}^0 \wedge ... \wedge \sigma_{n'}^0) \sqsubseteq \Gamma, (x : \sigma_1^0 \wedge ... \wedge \sigma_n^0)$. So, for every $\sigma' \sqsubseteq \sigma$, we have that there is some $\tau_d \sqsubseteq \sigma_0^0 \wedge ... \wedge \sigma_n^0$ such that $\tau_d \cong \sigma'$. So we can apply the induction hypothesis along with *T-Abs-1-f* or *T-Abs-0* to get that for every $\lambda x : \sigma'.E_f \lesssim \lambda x : \sigma.E$, there exists some $\lambda x : \tau_d.E_s \cong \lambda x : \sigma'.E_f$ with $\lambda x : \tau_d.E_s \sqsubseteq \lambda x : \sigma_1^0 \wedge ... \wedge \sigma_n^0.E'$, where if $\Gamma' \vdash \lambda x : \tau_d.E_s : \tau_d \to T_s$, then $\Gamma' \vdash_f \lambda x : \sigma'.E_f : \sigma' \to T'$ where $\tau_d \to T_s \sqsubseteq \sigma_0^0 \wedge ... \wedge \sigma_n^0 \to \tau'$ and $\sigma' \to T' \lesssim \sigma \to \tau$. We also have that $\sigma' \to T' \cong \tau_d \to T_s$ because $\lambda x : \tau_d.E_s \cong \lambda x : \sigma'.E_f$. We also have that $\Gamma' \sqsubseteq \Gamma$.

Case *T-Abs-2* is straightforward.

Case *T-App*

$$\frac{\begin{array}{cc} \Gamma \vdash_f E_1 : \sigma_1 & \Gamma \vdash_f E_2 : \sigma^0 \\ \sigma_1 \rhd ((\bigwedge_{i \in I} \tau_i^0) \to \sigma) & \forall i \leq I : \sigma^0 \sim \tau_i^0 \end{array}}{\Gamma \vdash_f E_1\ E_2 : \sigma} \textit{ (T-App)}$$

$$\frac{\begin{array}{cc}\Gamma \vdash E_{s_1} : \sigma_{s_1} & \Gamma \vdash E_{s_2} : \sigma^0 \\ \sigma_{s_1} \rhd ((\bigwedge_{i \in I} \tau^0_{s_i}) \to \sigma_s) & \forall i \leq I : \sigma^0_s \sim \tau^0_{s_i}\end{array}}{\Gamma \vdash E_{s_1}\ E_{s_2} : \sigma_s}\ \textit{(T-App)}$$

Since $\Gamma \vdash E_{s_1} : \sigma_{s_1}$ and $\Gamma \vdash_f E_1 : \sigma_1$ and $E_{s_1} \cong E_1$ and $\sigma_{s_1} \cong \sigma_1$, by induction, for every $E'_1 \lesssim E_1$, there exists some $E_{s'_1} \cong E'_1$ with $E_{s'_1} \sqsubseteq E_{s_1}$, where if $\Gamma' \vdash E_{s'_1} : \sigma_{s'_1}$, then $\Gamma' \vdash_f E'_1 : \sigma'_1$ where $\sigma'_1 \lesssim \sigma_1$, $\sigma_{s'_1} \sqsubseteq \sigma_{s_1}$ and $\Gamma' \sqsubseteq \Gamma$ and $\sigma_{s'_1} \cong \sigma'_1$.

Also since $\Gamma \vdash E_{s_2} : \sigma_{s_2}$ and $\Gamma \vdash_f E_2 : \sigma_2$ and $E_{s_2} \cong E_2$ and $\sigma_{s_2} \cong \sigma_2$, by induction, for every $E'_2 \lesssim E_2$, there exists some $E_{s'_2} \cong E'_2$ with $E_{s'_2} \sqsubseteq E_{s_2}$, where if $\Gamma' \vdash E_{s'_2} : \sigma_{s'_2}$, then $\Gamma' \vdash_f E'_2 : \sigma'_2$ where $\sigma'_2 \lesssim \sigma_2$, $\sigma_{s'_2} \sqsubseteq \sigma_{s_2}$ and $\Gamma' \sqsubseteq \Gamma$ and $\sigma_{s'_2} \cong \sigma'_2$.

Note that $\rhd$ is well-defined under $\cong$. So we have that $\sigma'_1 \rhd \bigwedge_{i \in I} \tau^0_{i'}) \to \sigma'$, and $\sigma'_{s_1} \rhd \bigwedge_{i \in I} \tau^0_{s'_i}) \to \sigma'_s$. Note that $\tau^0_{i'}$ maps to some $\tau^0_{s'_j}$ for $i, j \in 1, ..., n$ and that from $\sigma_{s'_2} \cong \sigma'_2$, we have that $\sigma_{s'_2} = \sigma'_2$. So we can apply *T-App* and the induction hypothesis to get that $\Gamma' \vdash_f E'_1\ E'_2 : \sigma'$ where $\sigma' \lesssim \sigma$, $\sigma'_s \sqsubseteq \sigma_s$ and $\Gamma' \sqsubseteq \Gamma$. Clearly, $\sigma' \cong \sigma'_s$ and $\Gamma' \sqsubseteq \Gamma$. □

**Theorem 4.7.1.** *If* $\Gamma \vdash_f E : T$ *then* $\exists T', E'$ *such that* $T' \cong T, E' \cong E$ *and* $\Gamma \vdash E' : T'$

*Proof.* We proceed by induction on the derivation.

Cases *T-True*, *T-False*, *T-Num*, *T-Var* are straightforward.

Case:

$$\frac{\begin{array}{l}\Gamma, (x : \sigma^0_0 \wedge \ldots \wedge \sigma^0_n) \vdash_f E : \tau \\ \textit{where } \forall i \in 0..n, x_i \textit{ occurs in } E \\ \textit{and } \sigma^0_0 = \ldots = \sigma^0_n = \sigma^0\end{array}}{\Gamma \vdash_f (\lambda x : \sigma^0.E) : \sigma^0 \to \tau}\ \textit{(T-Abs-0)}$$

By induction, we have that $\exists \tau', E'$ such that $\tau' \cong \tau, E' \cong E$ and $\Gamma, (x : \sigma_0^0 \wedge ... \wedge \sigma_n^0) \vdash E' : \tau'$.

Then we can apply:

$$\frac{\begin{array}{l}\Gamma, (x : \sigma_0^0 \wedge \ldots \wedge \sigma_n^0) \vdash E' : \tau' \\ \textit{where } \forall i \in 0..n, x_i \textit{ occurs in } E' \\ \textit{and } \sigma_0^0 = \ldots = \sigma_n^0 = \sigma^0\end{array}}{\Gamma \vdash (\lambda x : \sigma^0.E') : \sigma^0 \to \tau'} \textit{ (T-Abs-0)}$$

Case:

$$\frac{\begin{array}{l}\Gamma, (x : \sigma_0^0 \wedge \ldots \wedge \sigma_n^0) \vdash_f E : \tau \\ \textit{where } \forall i \in 0..n, x_i \textit{ occurs in } E \\ \textit{and } \sigma \cong \sigma_0^0 \wedge \ldots \wedge \sigma_n^0\end{array}}{\Gamma \vdash_f (\lambda x : \sigma.E) : \sigma \to \tau} \textit{ (T-Abs-1-f)}$$

By induction, we have that $\exists \tau', E'$ such that $\tau' \cong \tau, E' \cong E$ and $\Gamma, (x : \sigma_0^0 \wedge ... \wedge \sigma_n^0) \vdash E' : \tau'$.

So we have that:

$$\frac{\Gamma, (x : \sigma_0^0 \wedge ... \wedge \sigma_n^0) \vdash E' : \tau' \;\; \textit{where } \forall i \in 0..n, \; x_i \textit{ occurs in } E'}{\Gamma \vdash (\lambda x : \sigma_0^0 \wedge ... \wedge \sigma_n^0.E') : \sigma_0^0 \wedge ... \wedge \sigma_n^0 \to \tau'} \textit{ (T-Abs-1)}$$

Case *T-Abs-2* is straightforward.

Case

$$\frac{\begin{array}{cc}\Gamma \vdash_f E_1 : \sigma_1 & \Gamma \vdash_f E_2 : \sigma^0 \\ \sigma_1 \rhd ((\bigwedge_{i \in I} \tau_i^0) \to \sigma) & \forall i \leq I : \sigma^0 \sim \tau_i^0\end{array}}{\Gamma \vdash_f E_1\ E_2 : \sigma} \textit{ (T-App)}$$

By induction, we have that $\exists \sigma_1', E_1'$ such that $\sigma_1' \cong \sigma_1, E_1' \cong E_1$ and $\Gamma \vdash E_1' : \tau_1'$ and $\exists \sigma^{0'}, E_2'$ such that $\sigma^{0'} \cong \sigma^0, E_2' \cong E_2$ and $\Gamma \vdash E_2' : \sigma^{0'}$.

Note that from $\sigma^{0'} \cong \sigma^0$, we have that $\sigma^{0'} = \sigma^0$ and note that $\rhd$ is well-defined under $\cong$. So we have that $\sigma_1' \rhd ((\bigwedge_{i \in I} \tau_{i'}^0) \to \sigma')$ with $\sigma_1 \cong \sigma_1'$, $\sigma \cong \sigma'$ and $(\bigwedge_{i \in I} \tau_i^0) \cong (\bigwedge_{i \in I} \tau_{i'}^0)$.

Note that every $\tau_i^0$ maps to some $\tau_{j'}^0$ for $i, j \in 1, ..., n$.

So we have that:

$$\frac{\begin{array}{cc}\Gamma \vdash E_1' : \sigma_1' & \Gamma \vdash E_2' : \sigma^0 \\ \sigma_1' \rhd ((\bigwedge_{i \in I} \tau_{i'}^0) \to \sigma') & \forall i \leq I : \sigma^0 \sim \tau_{i'}^0\end{array}}{\Gamma \vdash E_1'\ E_2' : \sigma'} \textit{ (T-App)}$$

□

**Theorem 4.7.2.** *Suppose $\Gamma \vdash_f E : T$. Then for $E_s \cong E$ with $\Gamma \vdash E_s : T_s$, if $E_s \leq_\Gamma E_s'$, then there exists an $E'$ such that $E \leq_{\Gamma_f} E'$ with $E' \cong E_s'$.*

*Proof.* By expanding the definition of $\leq_\Gamma$, we have that some $\Gamma_s \vdash E_s' : T_s'$ with $E_s \sqsubseteq E_s'$, $T_s \sqsubseteq T_s'$ and $\Gamma \sqsubseteq \Gamma_s$. Since $E_s \cong E$ and $E_s \sqsubseteq E_s'$ then it must be the case that $E \lesssim E_s'$. Similarly, $\Gamma \lesssim \Gamma_s$ with $T \lesssim T_s'$. So we are done. □

# APPENDIX C

# Generalizing Shape Reasoning

## .1 Static Tensor types

$$
\begin{array}{rrcl}
(Program) & P & ::= & \texttt{decl}^{*}\ \texttt{return}\ e \\
(Decl)\ \texttt{decl} & & ::= & id : T \\
(Terms) & e & ::= & x \mid \texttt{add}(e_1, e_2) \mid \texttt{reshape}(e, T) \mid \\
 & & & \texttt{Conv2D}(c_{in}, c_{out}, \kappa_{kernel}, e) \\
(IntegerTuple) & \kappa & ::= & (c^{*}) \\
(Const) & c & ::= & \langle \texttt{Nat} \rangle \\
(Tensor\ Types) & S, T & ::= & \texttt{TensorType}(list(D)) \\
 & U, D & ::= & \langle \texttt{Nat} \rangle \\
(Env) & \Gamma & ::= & \emptyset \mid \Gamma, x : T
\end{array}
$$

$\delta$ denotes a tensor type with at most once `Dyn` dimension.

**Figure .1:** Tensor Calculus

Typing rules:

$$\frac{\texttt{decl}^* \vdash_s \Gamma \quad \Gamma \vdash_s e : T}{\Gamma \vdash_s \texttt{decl}^* \texttt{ return } e \texttt{ ok}} \textit{ (ok-prog-s)} \qquad \frac{x : T \in \Gamma}{\Gamma \vdash_s x : T} \textit{ (t-var)}$$

$$\frac{\Gamma \vdash_s e : \texttt{TensorType}(D_1, ..., D_n) \quad \prod_1^n D_i = \prod_1^m U_i}{\Gamma \vdash_s \texttt{reshape}(e, \texttt{TensorType}(U_1, ..., U_m)) : \texttt{TensorType}(U_1, ..., U_n)} \textit{ (t-reshape-s)}$$

$$\frac{\begin{array}{ll}\Gamma \vdash_s e : T, & T = \texttt{TensorType}(D_1, D_2, D_3, D_4), \\ S = \texttt{calc-conv}(T, c_{out}, \kappa_{kernel}), & c_{in} = D_2\end{array}}{\Gamma \vdash_s \texttt{Conv2D}(c_{in}, c_{out}, \kappa_{kernel}, e) : S} \textit{ (t-conv)}$$

$$\frac{\Gamma \vdash_s e_1 : T_1 \quad \Gamma \vdash_s e_1 : T_2 \quad (S_1, S_2) = \texttt{apply-broadcasting}(T_1, T_2) \quad S_1 = S_2}{\Gamma \vdash_s \texttt{add}(e_1 \ e_2) : S_1} \textit{ (t-add)}$$

**Figure .2:** Typing Rules

## .2 Gradual Tensor Types

## .3 Static properties

**Definition .1** (rank)**.** $\texttt{rank}(\texttt{TensorType}(d_1, ..., d_n)) = n$.

**Theorem .3.1** (Monotonicity w.r.t precision)**.** $\forall p, p', \Gamma$ : *if* $\Gamma \vdash p : \texttt{ok} \ \wedge \ p' \sqsubseteq p$ *then* $\Gamma \vdash p' : \texttt{ok}$.

*Proof.* Proof by induction on the proof structure of $p' \sqsubseteq p$.

Case $\texttt{decl}^{*'} \texttt{ return } e' \sqsubseteq \texttt{decl}^* \texttt{ return } e$. Then by inspection, we have:

$$
\begin{array}{rrcl}
(Program) & p & ::= & \texttt{decl}^*\ \texttt{return}\ e \\
(Decl)\ \texttt{decl} & & ::= & x : \tau \\
(Terms) & e & ::= & x \mid \texttt{add}(e_1, e_2) \mid \texttt{reshape}(e, \tau) \mid \\
 & & & \texttt{Conv2D}(c_{in}, c_{out}, \kappa_{kernel}, e) \\
(IntegerTuple) & \kappa & ::= & (c^*) \\
(Const) & c & ::= & \langle \texttt{Nat} \rangle \\
(Tensor\ Types) & t, \tau & ::= & \texttt{Dyn} \mid \texttt{TensorType}(list(d)) \\
(Static\ Tensor\ Types) & S, T & ::= & \texttt{TensorType}(list(D)) \\
 & \sigma, d & ::= & \texttt{Dyn} \mid \langle \texttt{Nat} \rangle \\
 & U, D & ::= & \langle \texttt{Nat} \rangle \\
(Env) & \Gamma & ::= & \emptyset \mid \Gamma, x : \tau
\end{array}
$$

Notation:
$\delta$ is a sequence of dimensions with at most one occurrence of `Dyn`.

**Figure .3:** Gradual Tensor Core language

$$
\frac{\forall i \in \{1, ..., n\}\ \texttt{decl}'_i \sqsubseteq \texttt{decl}_i \quad e' \sqsubseteq e}{\texttt{decl}'_1, ..., \texttt{decl}'_n\ \texttt{return}\ e' \sqsubseteq \texttt{decl}_1, ..., \texttt{decl}_n\ \texttt{return}\ e}\ \textit{(p-prog)}
$$

We also have the following rule:

$$
\frac{\texttt{decl}^* \vdash \Gamma \quad \Gamma \vdash e : \tau}{\Gamma \vdash \texttt{decl}^*\ \texttt{return}\ {}^*e\ \texttt{ok}}\ \textit{(ok-prog)}
$$

Consistency

$$\tau \sim \tau \ (c\text{-}refl\text{-}t) \quad d \sim d \ (c\text{-}refl\text{-}d)$$

$$d \sim \texttt{Dyn} \ (d\text{-}refl\text{-}dyn) \quad \tau \sim \texttt{Dyn} \ (t\text{-}refl\text{-}dyn)$$

$$\frac{\forall i \leq n : d_i \sim d'_i}{\texttt{TensorType}(d_1, ..., d_n) \sim \texttt{TensorType}(d'_1, ..., d'_n)} \ (c\text{-}tensor)$$

Type Precision

$$\tau \sqsubseteq \tau \ (refl\text{-}t) \quad d \sqsubseteq d \ (c\text{-}refl\text{-}d)$$

$$\texttt{Dyn} \sqsubseteq d \ (refl\text{-}dyn\text{-}1) \quad \texttt{Dyn} \sqsubseteq \tau \ (refl\text{-}dyn\text{-}2)$$

$$\frac{\forall i \leq n : d_i \sqsubseteq d'_i}{\texttt{TensorType}(d_1, ..., d_n) \sqsubseteq \texttt{TensorType}(d'_1, ..., d'_n)} \ (p\text{-}tensor)$$

Program and term Precision

$$\frac{\forall i \in \{1, ..., n\} \ \texttt{decl}'_i \sqsubseteq \texttt{decl}_i \quad e' \sqsubseteq e}{\texttt{decl}'_1, ..., \texttt{decl}'_n \ \texttt{return} \ e' \sqsubseteq \texttt{decl}_1, ..., \texttt{decl}_n \ \texttt{return} \ e} \ (p\text{-}prog) \quad \frac{\tau' \sqsubseteq \tau}{x : \tau' \sqsubseteq x : \tau} \ (p\text{-}decl)$$

$$e \sqsubseteq e \ (p\text{-}refl)$$

Matching

$$\texttt{TensorType}(\tau_1, ..., \tau_n) \rhd^n \texttt{TensorType}(\tau_1, ..., \tau_n)$$
$$\texttt{Dyn} \rhd^n \texttt{TensorType}(l) \text{ where } l = [\texttt{Dyn}, ..., \texttt{Dyn}] \text{ and } |l| = n$$

**Figure .4:** Auxiliary functions

$$\frac{}{\emptyset \vdash \bot}\ \textit{s-empty}$$

$$\frac{\texttt{decl}^* \vdash \Gamma \quad x \notin dom(\Gamma)}{\texttt{decl}^*\ id : \tau \vdash \Gamma : id \mapsto \tau}\ \textit{(s-var)}$$

**Figure .5:** Static context formation

We need to prove that $\Gamma' \vdash \texttt{decl}^{*'}\ \texttt{return}\ e'\ \texttt{ok}$.

We have that $\texttt{decl}^* \vdash \Gamma$. We consider $\texttt{decl}^{*'} \vdash \Gamma'$. Then we know that $\Gamma' \sqsubseteq \Gamma$.

Since $\Gamma \vdash e : \tau$, then by lemma .1, we have that $\Gamma' \vdash e' : \tau'$ where $\tau' \sqsubseteq \tau$. So we have that:

$$\frac{\texttt{decl}^{*'} \vdash \Gamma' \quad \Gamma' \vdash e' : \tau'}{\Gamma' \vdash \texttt{decl}^{*'}\ \texttt{return}\ e'\ \texttt{ok}}\ \textit{(ok-prog)}$$

□

**Lemma .1** (Monotonicity of expressions)**.** *Suppose* $\Gamma \vdash e : \tau$*. Then for* $\Gamma' \sqsubseteq \Gamma$ *and* $\Gamma' \vdash e : \tau'$ *with* $\tau' \sqsubseteq \tau$*.*

We proceed by induction on $e$.

Typing rules:

$$\frac{\texttt{decl}^* \vdash \Gamma \quad \Gamma \vdash e : \tau}{\vdash \texttt{decl}^* \ \texttt{return} \ e \ \texttt{ok}} \ \textit{(ok-prog)} \qquad \frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau} \ \textit{(t-var)}$$

$$\frac{\Gamma \vdash e : \texttt{TensorType}(D_1, ..., D_n) \quad \prod_1^n D_i = \prod_1^m U_i}{\Gamma \vdash \texttt{reshape}(e, \texttt{TensorType}(U_1, ..., U_m)) : \texttt{TensorType}(U_1, ..., U_m)} \ \textit{(t-reshape-s)}$$

$$\frac{\begin{array}{c}\Gamma \vdash e : \texttt{TensorType}(\sigma_1, ..., \sigma_m) \\ \prod_1^m \sigma_i \ mod \ \prod_1^n d_i = 0 \vee \prod_1^n d_i \ mod \ \prod_1^m \sigma_i = 0 \ \forall d_i, \sigma_i \neq \texttt{Dyn} \\ \textit{and } \texttt{Dyn} \textit{ occurs exactly once in } d_1, ...., d_m, \sigma_1, ..., \sigma_n \\ \textit{or} \\ \texttt{Dyn} \textit{ occurs more than once in } d_1, ...., d_m,\end{array}}{\Gamma \vdash \texttt{reshape}(e, \texttt{TensorType}(d_1, ..., d_n)) : \texttt{TensorType}(d_1, ..., d_n)} \ \textit{(t-reshape-g)}$$

$$\frac{\begin{array}{c}\Gamma \vdash e : \tau \ \textit{where} \\ \tau = \texttt{TensorType}(\sigma_1 \ ... \ \sigma_n) \\ \textit{and } \texttt{Dyn} \textit{ occurs more than once with at least one occurrence in} \\ \delta \ \textit{and } \sigma_1, ...., \sigma_m \\ \textit{or } \tau = \texttt{Dyn}\end{array}}{\Gamma \vdash \texttt{reshape}(e, \delta) : \delta} \ \textit{(t-reshape)}$$

$$\frac{\Gamma \vdash e : t \quad t \rhd^4 \texttt{TensorType}(\sigma_1, \sigma_2, \sigma_3, \sigma_4) \quad \tau = \texttt{calc-conv}(t, c_{out}, \kappa_{kernel}) \quad c_{in} \sim \sigma_2}{\Gamma \vdash \texttt{Conv2D}(c_{in}, c_{out}, \kappa_{kernel}, e) : \tau} \ \textit{(t-conv)}$$

$$\frac{\Gamma \vdash e_1 : t_1 \quad \Gamma \vdash e_1 : t_2 \quad (\tau_1, \tau_2) = \texttt{apply-broadcasting}(t_1, t_2) \quad \tau_1 \sim \tau_2}{\Gamma \vdash \texttt{add}(e_1, e_2) : \tau_1 \sqcap^* \tau_2} \ \textit{(t-add)}$$

**Figure .6:** Typing Rules

Case $x$.

We clearly have that $\Gamma \vdash x : \tau$ and $\Gamma' \vdash x : \tau'$ and $\tau' \sqsubseteq \tau$.

**Greatest lower bound:**

$$\tau \sqcap \tau' | \tau \nsim \tau' = \texttt{undefined}$$
$$\tau \sqcap \tau = \tau$$
$$\texttt{Dyn} \sqcap \tau = \tau$$
$$\tau \sqcap \texttt{Dyn} = \tau$$
$$\texttt{TensorType}(d_1, ..., d_n) \sqcap \texttt{TensorType}(d'_1, ..., d'_n) = \texttt{TensorType}(d_1 \sqcap d'_1, ..., d_n \sqcap d'_n)$$
$$d_1 \sqcap d_1 = d_1$$
$$d_1 \sqcap \texttt{Dyn} = d_1$$
$$\texttt{Dyn} \sqcap d_2 = d_2$$
$$d_1 \sqcap d_2 | d_1 \nsim d_2 = \texttt{undefined}$$

**Greatest lower bound *:**

$$\tau \sqcap^* \tau' | \tau \nsim \tau' = \texttt{undefined}$$
$$\texttt{Dyn} \sqcap^* \tau = \texttt{Dyn}$$
$$\tau \sqcap^* \texttt{Dyn} = \texttt{Dyn}$$
$$\texttt{TensorType}(d_1, ..., d_n) \sqcap^* \texttt{TensorType}(d'_1, ..., d'_n) =$$
$$\texttt{TensorType}(d_1, ..., d_n) \sqcap \texttt{TensorType}(d'_1, ..., d'_n)$$

**apply-broadcasting:**

*apply-broadcasting*$(\tau_1, \tau_2)$ *is defined in the following way:*
If $\tau_1 = \texttt{Dyn} \vee \tau_2 = \texttt{Dyn}$, then return $\tau_1, \tau_2$.
Otherwise:

- Let $\tau_1$ and $\tau_2$ be equal in length by padding the shorter type with 1's from index 0.
- Replace occurrences of 1 in $\tau_1$ with the type at the same index in $\tau_2$.
- Replace occurrences of 1 in $\tau_2$ with the type at the same index in $\tau_1$.

**calc-conv:**

Let $t \triangleright \texttt{TensorType}(\sigma_0, \sigma_1, \sigma_2, \sigma_3)$. Then
$\texttt{calc} - \texttt{conv}(t, c_{out}, \kappa_{kernel}) = \texttt{TensorType}(t'_0, t'_1, t'_2, t'_3)$ where:

$$t'_0 = \sigma_0$$
$$t'_1 = c_{out}$$
$$t'_2 = \begin{cases} \sigma_2 - 1 \times (\kappa_{kernel}[0] - 1) & \text{if } \sigma_2 \in \mathbb{N} \\ \texttt{Dyn} & \text{otherwise} \end{cases}$$
$$t'_3 = \begin{cases} \sigma_3 - 1 \times (\kappa_{kernel}[1] - 1) & \text{if } \sigma_3 \in \mathbb{N} \\ \texttt{Dyn} & \text{otherwise} \end{cases}$$



**Figure .7:** Broadcasting, convolution, and greatest lower bound

If two tensors $x$, $y$ are "broadcastable", let $broadcast(x, y)$ denote the resulting tensor shapes. The resulting tensor shapes are calculated as follows:

1. If the number of dimensions of $x$ and $y$ are not equal, prepend 1 to the dimensions of the tensor with fewer dimensions to make them equal length.
2. Then, for each dimension size, the resulting dimension size is the max of the shapes of $x$ and $y$ along that dimension.

**Figure .8:** Broadcasting runtime semantics

Case $\texttt{add}(e_1, e_2)$

We have that:

$$\frac{\Gamma \vdash e_1 : t_1 \quad \Gamma \vdash e_1 : t_2 \quad (\tau_2, \tau_2) = \texttt{apply-broadcasting}(t_1, t_2) \quad \tau_1 \sim \tau_2}{\Gamma \vdash \texttt{add}(e_1, e_2) : \tau_1 \sqcup^* \tau_2} \textit{ (t-add)}$$

By applying the IH, we have that $\Gamma' \vdash e_1 : t_1'$ and $\Gamma' \vdash e_2 : t_2'$ where $t_1' \sqsubseteq t_1$ and $t_2' \sqsubseteq t_2$. Note that `apply-broadcasting` preserves monotonicity, by lemma .2. Furthermore, $\sqcup^*$ and $\sim$ preserve monotonicity. Therefore we can apply *(t-add)* again to get that $\Gamma' \vdash \texttt{add}(e_1, e_2) : t'$ where $t' \sqsubseteq t$.

Case $\texttt{reshape}(e, \tau)$.

We will proceed with case analysis on the derivation rules.

Consider:

$$\frac{\Gamma \vdash e : \texttt{TensorType}(D_1, ..., D_n) \quad \prod_1^n D_i = \prod_1^m U_i}{\Gamma \vdash \texttt{reshape}(e, \texttt{TensorType}(U_1, ..., U_m)) : \texttt{TensorType}(U_1, ..., U_n)} \textit{ (t-reshape-s)}$$

.

By applying the IH, we have that $\Gamma' \vdash e : t$ where $t \sqsubseteq \texttt{TensorType}(D_1, ..., D_n)$. First, if $t = \texttt{Dyn}$ or has more than one occurrence of `Dyn` then we can either *t-reshape* or *t-reshape-g* depending on the occurrences to get that $\Gamma' \vdash \texttt{reshape}(e, \tau) : \tau$. If $t = \texttt{TensorType}(U_1, ..., U_n)$ then it must be the case that $D_1 = U_1, ..., D_n = U_n$. Otherwise, we know that $\prod_1^n D_i = \prod_1^m U_i$ and that $\tau'$ is the same as $\tau$ except that one dimension is replaced with `Dyn`. Therefore, $\prod_1^n D_i$ is divisible by the product of dimensions of $\tau'$ so we can apply *t-reshape-g* or *t-reshape* depending on the `Dyn` occurrences.

Next, consider:

$$\frac{\begin{array}{c}\Gamma \vdash e : \texttt{TensorType}(\sigma_1, ..., \sigma_m) \\ \prod_1^m \sigma_i \ mod \ \prod_1^n d_i = 0 \vee \prod_1^n d_i \ mod \ \prod_1^m \sigma_i = 0 \ \forall d_i, \sigma_i \neq \texttt{Dyn} \\ \textit{and } \texttt{Dyn} \textit{ occurs exactly once in } d_1, ...., d_m, \sigma_1, ..., \sigma_n \\ \textit{or} \\ \texttt{Dyn} \textit{ occurs more than once in } d_1, ...., d_m,\end{array}}{\Gamma \vdash \texttt{reshape}(e, \texttt{TensorType}(d_1, ..., d_n)) : \texttt{TensorType}(d_1, ..., d_n)} \textit{ (t-reshape-g)}$$

From the IH, we have that $\Gamma \vdash e : t$ with $t \sqsubseteq \texttt{TensorType}(\sigma_1, ..., \sigma_m)$. Consider $t$. If $t = \texttt{TensorType}(\sigma_1, ..., \sigma_m)$ then apply *t-reshape-g* or *t-resshape* depending on the `Dyn` occurrences

Finally, we consider:

$$\frac{\begin{array}{c}\Gamma \vdash e : \tau \textit{ where} \\ \tau = \texttt{TensorType}(\sigma_1 \ ... \ \sigma_n) \\ \textit{and } \texttt{Dyn} \textit{ occurs more than once with at least one occurrence in} \\ \delta \textit{ and } \sigma_1, ...., \sigma_m \\ \textit{or } \tau = \texttt{Dyn}\end{array}}{\Gamma \vdash \texttt{reshape}(e, \delta) : \delta} \textit{ (t-reshape)}$$

Then by the IH. we have that $\Gamma' \vdash e : t$ where $t \sqsubseteq \tau$. In this case, we will apply *t-reshape*.

Case $\texttt{Conv2D}(c_{in}, c_{out}, \kappa_{kernel}, e)$.

Then we have:

$$\frac{\Gamma \vdash e : t \quad t \rhd^4 \texttt{TensorType}(\sigma_1, \sigma_2, \sigma_3, \sigma_4) \quad \tau = \texttt{calc-conv}(t, c_{out}, \kappa_{kernel}) \quad c_{in} \sim \sigma_2}{\Gamma \vdash \texttt{Conv2D}(c_{in}, c_{out}, \kappa_{kernel}, e) : \tau} \textit{ (t-conv)}$$

From the IH, $\Gamma' \vdash e' : t'$ with $t' \sqsubseteq t$ and $e' \sqsubseteq e$. We know that $t' \rhd^4 (\sigma'_1, \sigma'_2, \sigma'_3, \sigma'_4)$ with $\sigma'_i \sqsubseteq \sigma_i$ for $i \in \{1, ..., 4\}$. Since `calc-conv` preserves monotonicity, by lemma .3, then $calc - conv(t', c_{out}, \kappa_{kernel}) = \tau'$ for $\tau' \sqsubseteq \tau$ so we can apply *t-conv* and we are done.

**Lemma .2** (Monotonicity of broadcasting)**.** *for $t'_1 \sqsubseteq t_1$ and $t'_2 \sqsubseteq t_2$, we have that if* `apply-broadcasting`$(t_1, t_2) = \tau_1, \tau_2$ *then* `apply-broadcasting`$(t'_1, t'_2) = \tau'_1, \tau'_2$ *where $\tau'_1 \sqsubseteq \tau_1$ and $\tau'_2 \sqsubseteq \tau_2$.*

*Proof.* If either $t_1 = \texttt{Dyn}$ or $t_2 = \texttt{Dyn}$ then we return $t_1$ and $t_2$. By the definition of precision, we must have that either either $t_1' = \texttt{Dyn}$ or $t_2' = \texttt{Dyn}$ then we return $t_1'$ and $t_2'$ and we already know that $t_1' \sqsubseteq t_1$ and $t_2' \sqsubseteq t_2'$ so we are done.

Otherwise, we know that $t_1, t_2, t_1'$ and $t_2'$ are tensor types.
Consider $\texttt{apply-broadcasting}(t_1, t_2) = \tau_1, \tau_2$ and $\texttt{apply-broadcasting}(t_1', t_2') = \tau_1', \tau_2'$. We know that $t_1 \sim t_1'$ and $t_2 \sim t_2'$. So $\texttt{rank}(t_1) = \texttt{rank}(t_1')$ and $\texttt{rank}(t_2) = \texttt{rank}(t_2')$. Broadcasting preserves length. Therefore, $\texttt{rank}(\tau_1) = \texttt{rank}(\tau_1')$ and $\texttt{rank}(\tau_2) = \texttt{rank}(\tau_2')$.

Now we must show that each of the elements are related by precision, so let $t_1 = \texttt{TensorType}(d_1, ..., d_n), t_1' = \texttt{TensorType}(d_1', ..., d_n'), t_2 = \texttt{TensorType}(k_1, ..., k_n), t_2' = \texttt{TensorType}(k_1', ..., k_n')$.
Then we will have $\tau_1 = \texttt{TensorType}(\delta_1, ..., \delta_n), \tau_1' = \texttt{TensorType}(\delta_1', ..., \delta_n'), \tau_2 = \texttt{TensorType}(\kappa_1, ..., \kappa_n), \tau_2' = \texttt{TensorType}(\kappa_1', ..., \kappa_n')$.

Assume $d_i = 1$ then $\delta_i = k_i$ and $d_i' = 1$ so $\delta_i' = k_i'$ and we know that $k_i' \sqsubseteq k_i$. Similarly, if $k_i = 1$ then $\kappa_i = d_i$ and $k_i' = 1$ so $\kappa_i' = d_i'$ and we have that $d_i' \sqsubseteq d_i$. □

**Lemma .3** (Monotonicity of convolution)**.** *for tensor types $t', t$ :*
*if $t' \sqsubseteq t$ and $\texttt{calc-conv}(t, c_{out}, \kappa_{kernel}) = \tau$ then $\texttt{calc-conv}(t', c_{out}, \kappa_{kernel}) = \tau'$ where $\tau' \sqsubseteq \tau$.*

*Proof.* Consider $t = \texttt{TensorType}(d_1, ..., d_n)$ and $t' = \texttt{TensorType}(d_1', ..., d_n')$. By applying `calc-conv`, we have that $d_1 = d_1'$ and $d_2 = d_2'$. By inspection, $d_3' \sqsubseteq d_3$ and $d_4' \sqsubseteq d_4$. □

**Lemma .4** (Monotonicity of matching)**.** *if $t_1' \rhd^i t_2'$ and $t_1' \sqsubseteq t_1$ then $t_1 \rhd^i t_2$ and $t_2' \sqsubseteq t_2$.*

*Proof.* Straightforward. □

**Theorem .5.** *Let $\tau_1 \sim \tau_2$. Then $\exists \tau_3$ such that $\tau_1 \sqcup^* \tau_2 = \tau_3$*

*Proof.* We proceed by induction on the derivation.

Consider $\tau \sim \tau$ (*c-refl-t*). Then $\tau \sqcup^* \tau = \tau$. Next, consider $\tau \sim \texttt{Dyn}$. Then we have that $\tau \sqcup^* \texttt{Dyn} = \texttt{Dyn}$.

Next, consider

$$\frac{\forall i \leq n : \tau_i \sim \tau_i'}{\texttt{TensorType}(\tau_1, ..., \tau_n) \sim \texttt{TensorType}(\tau_1', ..., \tau_n')} \ (c\text{-}tensor)$$

Then by induction, we have that $\forall i \in \{1, ..., n\} : \tau_i' \sim \tau_i$ so we have that $\tau_i' \sqcup^* \tau_i = \tau_i$". Then we get that $\texttt{TensorType}(\tau_1, ..., \tau_n) \sqcup^* \texttt{TensorType}(\tau_1', ..., \tau_n') = \texttt{TensorType}(\tau_1", ..., \tau_n")$ □

**Theorem .6.** *Gradual Tensor Types are unique*

*Proof.* Straightforward. □

**Theorem .7** (Conservative Extension)**.** *For all static* $\Gamma, p$*, we have:*
$\Gamma \vdash_S p : \texttt{ok}$ *iff* $\Gamma \vdash p : \texttt{ok}$

**Forward direction.** We proceed by induction on derivation.

*Proof.* Case *ok-prog-s*

$$\frac{\texttt{decl}^* \vdash_s \Gamma \quad \Gamma \vdash_s e : T}{\Gamma \vdash_s \texttt{decl}^* \ \texttt{return} \ e \ \texttt{ok}} \ \textit{(ok-prog-s)}$$

so clearly:

$$\frac{\texttt{decl}^* \vdash \Gamma \quad \Gamma \vdash e : T}{\Gamma \vdash \texttt{decl}^* \ \texttt{return} \ e \ \texttt{ok}} \ \textit{(ok-prog)}$$

Case *t-var* is straightforward.

Case *t-reshape-s* maps directly to a rule in the gradual language so it is also straightfoward.

case *t-conv*

$$\frac{\begin{array}{ll}\Gamma \vdash_s e : T, & T = \texttt{TensorType}(D_1, D_2, D_3, D_4), \\ S = \texttt{calc-conv}(T, c_{out}, \kappa_{kernel}), & c_{in} = D_2\end{array}}{\Gamma \vdash_s \texttt{Conv2D}(c_{in}, c_{out}, \kappa_{kernel}, e) : S} \textit{ (t-conv)}$$

So we have:

$$\frac{\begin{array}{l}\Gamma \vdash e : t, \quad T \rhd^4 \texttt{TensorType}(D_1, D_2, D_3, D_4), \quad T = \texttt{calc-conv}(T, c_{out}, \kappa_{kernel}), \\ \qquad\qquad c_{in} \sim \sigma_2\end{array}}{\Gamma \vdash \texttt{Conv2D}(c_{in}, c_{out}, \kappa_{kernel}, e) : T} \textit{ (t-conv)}$$

Similarly for:

$$\frac{\Gamma \vdash_s e_1 : T_1 \;\; \Gamma \vdash_s e_1 : T_2 \;\; (S_1, S_2) = \texttt{apply-broadcasting}(T_1, T_2) \;\; S_1 = S_2}{\Gamma \vdash_s \texttt{add}(e_1 \; e_2) : S_1} \textit{ (t-add)}$$

we have:

$$\frac{\Gamma \vdash e_1 : S_1 \;\; \Gamma \vdash e_1 : S_2 \;\; (S_1, S_2) = \texttt{apply-broadcasting}(S_1, S_2) \;\; S_1 \sim S_2}{\Gamma \vdash \texttt{add}(e_1, e_2) : S_1 \sqcup^* S_2} \textit{ (t-add)}$$

Here, note that since $S_1$ and $S_2$ are static and $S_1 = S_2$ then $S_1 \sqcup^* S_2 = S_1$

**Backwards direction.** We can proceed by induction on the derivation. We have:

$$\frac{\texttt{decl}^* \vdash \Gamma \quad \Gamma \vdash e : T}{\Gamma \vdash \texttt{decl}^* \ \texttt{return} \ e \ \texttt{ok}} \ \textit{(ok-prog)}$$

From $\texttt{decl}^* \vdash \Gamma$, we get that $\texttt{decl}^* \vdash_s \Gamma$.

From the induction on the sub derivation, we get that $\Gamma \vdash_s e : T$. Therefore, :

$$\frac{\texttt{decl}^* \vdash_s \Gamma \quad \Gamma \vdash_s e : T}{\Gamma \vdash_s \texttt{decl}^* \ \texttt{return} \ e \ \texttt{ok}} \ \textit{(ok-prog)}$$

*t-var* is straightforward.

*t-reshape-g* and *t-reshape* do not apply since they all involve the `Dyn` type.

For *t-reshape-s* we get:

$$\frac{\Gamma \vdash e : \texttt{TensorType}(D_1, ..., D_n) \quad \prod_1^n D_i = \prod_1^m U_i}{\Gamma \vdash \texttt{reshape}(e, \texttt{TensorType}(U_1, ..., U_m)) : \texttt{TensorType}(U_1, ..., U_n)} \ \textit{(t-reshape-s)}$$

.

we can apply the IH and get that $\Gamma \vdash_s e : \texttt{TensorType}(D_1, ..., D_n)$. Therefore:

$$\frac{\Gamma \vdash_s e : \texttt{TensorType}(D_1, ..., D_n) \quad \prod_1^n D_i = \prod_1^m U_i}{\Gamma \vdash_s \texttt{reshape}(e, \texttt{TensorType}(U_1, ..., U_m)) : \texttt{TensorType}(U_1, ..., U_n)} \ \textit{(t-reshape-s)}$$

.

For *t-conv* we get:

$$\frac{\begin{array}{c}\Gamma \vdash e : S \\ S \rhd^4 \texttt{TensorType}(D_1, D_2, D_3, D_4) \\ T = \texttt{calc-conv}(t, c_{out}, \kappa_{kernel}) \\ c_{in} \sim D_2\end{array}}{\Gamma \vdash \texttt{Conv2D}(c_{in}, c_{out}, \kappa_{kernel}, e) : T} \textit{ (t-conv)}$$

from the IH, we get that $\Gamma \vdash_s e : S$. We know that $\rightarrow$ and $\sim$ are equality on static types, so we can directly apply *t-conv* to get

$$\frac{\begin{array}{c}\Gamma \vdash_s e : S \\ S = \texttt{TensorType}(D_1, D_2, D_3, D_4) \\ T = \texttt{calc-conv}(t, c_{out}, \kappa_{kernel}) \\ c_{in} = D_2\end{array}}{\Gamma \vdash_s \texttt{Conv2D}(c_{in}, c_{out}, \kappa_{kernel}, e) : T} \textit{ (t-conv)}$$

Next, we have:

$$\frac{\Gamma \vdash e_1 : S_1 \;\; \Gamma \vdash e_2 : S_2 \;\; (T_2, T_2) = \texttt{apply-broadcasting}(S_1, S_2) \;\; T_1 \sim T_2}{\Gamma \vdash \texttt{add}(e_1, e_2) : T_1 \sqcup^* T_2} \textit{ (t-add)}$$

We have that $\Gamma \vdash_s e_1 : T_1$ and $\Gamma \vdash_s e_2 : T_2$. We know that $T_1 \sim T_2$ so $T_1 = T_2$. Therefore, $T_1 \sqcup^* T_2 = T_1$ so we get:

$$\frac{\Gamma \vdash_s e_1 : S_1 \;\; \Gamma \vdash_s e_2 : S_2 \;\; (T_2, T_2) = \texttt{apply-broadcasting}(S_1, S_2) \;\; T_1 = T_2}{\Gamma \vdash_s \texttt{add}(e_1, e_2) : T_1} \; \textit{(t-add)}$$

□

$$\frac{\texttt{decl}^* \vdash \Gamma \;\; \texttt{decl}^* \vdash_d \mu \;\; \texttt{eval}_{\Gamma,\mu}(e) = v}{\texttt{decl}^* \; \texttt{return} \; e \Downarrow v} \; \texttt{(ev-prog)}$$

## .4 Type Migration

From $e, \Gamma$, we generate constraints $Gen(e, \Gamma)$ as follows. Assume that $e$ has been $\alpha$-converted so that all bound variables are distinct from each other and distinct from the free variables. Let $X$ be the set of declaration-variables $x$ occurring in $e$, and let $Y$ be a set of variables disjoint from $X$ consisting of a variable $[\![e']\!]$ for every occurrence of the subterm $e'$ in $e$. Let $Z$ be a set of variables disjoint from $X$ and $Y$ consisting of a variable $\langle e_1 \rangle, \langle e_2 \rangle$ for every occurrence of the subterm $\texttt{add}(e_1, e_2)$ in $e$. Finally, let $V$ be a set of variables disjoint from $X$, $Y$ and $Z$ consisting of dimension variables $\zeta$. The notations $[\![e]\!]$ and $\langle e \rangle$ are ambiguous because there may be more than one occurrence of some subterm $e'$ in $e$ or some subterm $e''$

in $e$. However, it will always be clear from context which occurrence is meant. We will Now we generate the following constraints.

$$
\begin{aligned}
(\mathit{Constraints}) \quad \psi \quad ::= \quad & \psi \wedge \psi \mid \psi \vee \psi \mid \texttt{True} \mid \texttt{False} \\
& \mid [\![x]\!] = x \mid [\![e]\!] = \tau \mid \tau \sqsubseteq x \\
& \mid |[\![e]\!]| \leq 5 \mid D \sim \delta \mid \langle e \rangle \sim \langle e \rangle \\
& \mid [\![e]\!] \rhd \texttt{TensorType}(\zeta_1, \zeta_2, \zeta_3, \zeta_4) \\
& \mid [\![e]\!] = \langle e \rangle \sqcup^* \langle e \rangle \mid \texttt{can-reshape}([\![e]\!], \delta) \\
& \mid [\![e]\!] = \texttt{calc-conv}([\![e]\!], c_{out}, \kappa_{kernel}) \\
& \mid \langle e \rangle, \langle e \rangle = \texttt{apply-broadcasting}([\![e]\!], [\![e]\!])
\end{aligned}
$$

Constraint generation:

$$\frac{}{\Gamma \vdash x : \tau : \tau \sqsubseteq x \wedge |x| \leq 5}\ (t\text{-}decl) \qquad \frac{}{\Gamma \vdash x : x = [\![x]\!]}\ (t\text{-}var)$$

$$\frac{\Gamma \vdash e : \psi}{\begin{array}{c}\Gamma \vdash \texttt{reshape}(e, \delta) : \psi \cup\ [\![\texttt{reshape}(e, \delta)]\!] = \delta \wedge \\ \texttt{can-reshape}([\![e]\!], \delta) \wedge |[\![e]\!]| \leq 5\end{array}}\ (t\text{-}reshape)$$

$$\frac{\Gamma \vdash e : \psi}{\begin{array}{c}\Gamma \vdash \texttt{Conv2D}(c_{in}, c_{out}, \kappa_{kernel}, e) : \psi \cup \\ [\![e]\!] \rhd \texttt{TensorType}(\zeta_1, \zeta_2, \zeta_3, \zeta_4) \wedge c_{in} \sim \zeta_2 \wedge \\ [\![\texttt{Conv2D}(c_{in}, c_{out}, \kappa_{kernel}, e)]\!] = \texttt{calc-conv}([\![e]\!], c_{out}, \kappa_{kernel})\end{array}}\ (t\text{-}conv)$$

$$\frac{\Gamma \vdash e_1 : \psi_1 \quad \Gamma \vdash e_2 : \psi_2}{\begin{array}{c}\Gamma \vdash \texttt{add}(e_1, e_2) : \psi_1 \cup \psi_2 \cup [\![\texttt{add}(e_1, e_2)]\!] = \langle e_1 \rangle \sqcup^* \langle e_2 \rangle \wedge \\ (\langle e_1 \rangle, \langle e_2 \rangle) = \texttt{apply-broadcasting}([\![e_1]\!], [\![e_2]\!]) \wedge \langle e_1 \rangle \sim \langle e_2 \rangle \wedge \\ |[\![e_1]\!]| \leq 5 \wedge |[\![e_2]\!]| \leq 5 \wedge |[\![\texttt{add}(e_1, e_2)]\!]| \leq 5\end{array}}\ (t\text{-}add)$$

**Figure .9:** Constraint generation

### .4.1 Constraint resolution

First, we will describe the grammar of the resulting constraints. We define $\texttt{IntConst} = \mathbb{Z}^+$ and we use $n$ to range over $\texttt{IntConst}$. We use $v$ as a metavariable that ranges over variables that range over $\texttt{TensorType}(list(\zeta)) \cup \{Dyn\}$, and we use $\zeta$ as a metavariable that ranges over variables that range over $\texttt{IntConst} \cup \{Dyn\}$.

$$
\begin{aligned}
(\textit{Constraints})\quad \psi \quad ::= \quad & \psi \wedge \psi \mid \psi \vee \psi \mid \neg\psi \mid \texttt{True} \mid \texttt{False} \\
& \mid v = \texttt{TensorType}(\zeta, ..., \zeta) \mid v = \texttt{Dyn} \mid v = v \\
& \mid \zeta = n \mid \zeta = \texttt{Dyn} \mid \zeta = \zeta \\
& \mid \zeta = \zeta * n + n \\
& \mid (\zeta_1 * ... * \zeta_m) \; mod \; (\zeta'_1 * ... * \zeta'_n) = 0
\end{aligned}
$$

We define a solution $\psi$ as follows.

| For each: | we have: |
|---|---|
| $\psi \wedge \psi'$ | $\varphi \vDash \psi \wedge \psi'$ |
| $\psi \vee \psi'$ | $\varphi \vDash \psi \vee \psi'$ |
| $\neg\psi$ | $\varphi \vDash \neg\psi$ |
| $\texttt{True}$ | $\varphi \vDash \texttt{True}$ |
| $\texttt{False}$ | $\varphi \vDash \texttt{False}$ |
| $v = \texttt{TensorType}(\zeta_1, ...\zeta_n)$ | $\varphi(v) = \texttt{TensorType}(\varphi(\zeta_1), ...\varphi(\zeta_n))$ |
| $v = \texttt{Dyn}$ | $\varphi(v) = \texttt{Dyn}$ |
| $v = v'$ | $\varphi(v) = \varphi(v')$ |
| $\zeta = n$ | $\varphi(\zeta) = n$ |
| $\zeta = \texttt{Dyn}$ | $\varphi(\zeta) = \texttt{Dyn}$ |
| $\zeta = \zeta'$ | $\varphi(\zeta) = \varphi(\zeta')$ |
| $\zeta = \zeta * n + n'$ | $\varphi(\zeta) = \varphi(\zeta') * n + n'$ |
| $(\zeta_1 * ... * \zeta_m) \; mod \; (\zeta'_1 * ... * \zeta'_n) = 0$ | $(\varphi(\zeta_1) * ... * \varphi(\zeta_m)) \; mod \; (\varphi(\zeta'_1) * ... * \varphi(\zeta'_n)) = 0$ |

Next, we define a series of steps for constraint resolution.

**Precision constraints.** We define a simplification procedure *SimPrec* that transforms every Precision constraint into zero, one, or more Equality constraints:

We define *SimPrec* to leave the set of type variables unchanged, and to proceed by repeating the following transformation until it no longer has an effect.

| From | To |
|---|---|
| $\texttt{Dyn} \sqsubseteq x$ | (no constraint) |
| $\texttt{TensorType}(D_1, ..., D_n) \sqsubseteq x$ | $x = \texttt{TensorType}(D_1, ..., D_n)$ |
| $\texttt{TensorType}(d_1, ..., d_n) \sqsubseteq x$ | $x = \texttt{TensorType}(\zeta_1, ..., \zeta_n) \ \wedge \ \forall i \in \{1, ..., n\} : d_i \sqsubseteq \zeta_i$ where $\zeta_1, ..., \zeta_n$ are fresh type variables |
| $D \sqsubseteq \zeta$ | $D = \zeta$ |
| $\texttt{Dyn} \sqsubseteq \zeta$ | (no constraint) |

**Matching constraints.** We define a simplification procedure *SimMatch* that replaces each Matching constraint with Equality and Disjunction constraints.

$$From : [\![e]\!] \rhd \texttt{TensorType}(\zeta_1, \zeta_2, \zeta_3, \zeta_4)$$

$$To : ([\![e]\!] = \texttt{Dyn} \wedge \zeta_1 = \texttt{Dyn} \wedge \zeta_2 = \texttt{Dyn} \wedge \zeta_3 = \texttt{Dyn} \wedge \zeta_4 = \texttt{Dyn}) \vee ([\![e]\!] = \texttt{TensorType}(\zeta_1, \zeta_2, \zeta_3, \zeta_4)$$

$\leq$ **constraints.** We define a simplification procedure *SimLessThan* that replaces each Matching constraint with Equality and Disjunction constraints.

$$From : |[\![e]\!]| \leq 4$$

$$To : [\![e]\!] = \texttt{Dyn} \vee [\![e]\!] = \texttt{TensorType}(\zeta_1) \vee ... \vee [\![e]\!] = \texttt{TensorType}(\zeta_1, \zeta_2, \zeta_3, \zeta_4)$$

where $\zeta_1, ..., \zeta_4$ are fresh variables

**Convolution constraints.**

$$[\![e]\!] = \texttt{calc-conv}([\![e']\!], c_{out}, \kappa_{kernel})$$

First, from a previous constraint, we know that $[\![e']\!] \rhd \texttt{TensorType}(\zeta_1, \zeta_2, \zeta_3, \zeta_4)$

$$From : [\![e]\!] = \texttt{calc-conv}([\![e']\!], c_{out}, \kappa_{kernel})$$

$$
\begin{aligned}
To : [\![e]\!] = \texttt{TensorType}(\epsilon_1, \epsilon_2, \epsilon_3, \epsilon_4) \wedge \\
\epsilon_1 = \zeta_1 \wedge \\
\epsilon_2 = c_{out} \wedge \\
(((\epsilon_3 = \texttt{Dyn} \wedge \\
\zeta_3 = \texttt{Dyn}) \vee \\
(\zeta_3 \neq \texttt{Dyn} \wedge \\
\epsilon_3 = (\zeta_3 - 1 \times (\kappa_{kernel}[0] - 1) - 1) + 1)) \wedge \\
(\epsilon_4 = \texttt{Dyn} \wedge \\
\zeta_4 = \texttt{Dyn}) \vee \\
(\zeta_4 \neq \texttt{Dyn} \wedge \\
\epsilon_4 = (\zeta_4 - 1 \times (\kappa_{kernel}[0] - 1) - 1) + 1))
\end{aligned}
$$

**Broadcasting constraints.**

$$
From : \langle e_1 \rangle, \langle e_2 \rangle = \texttt{apply-broadcasting}([\![e_1]\!], [\![e_2]\!])
$$

$$
\begin{aligned}
&[\![e_1]\!] = \texttt{Dyn} \wedge \langle e_1 \rangle = [\![e_1]\!] \wedge \langle e_2 \rangle = [\![e_2]\!] \vee \\
&[\![e_2]\!] = \texttt{Dyn} \wedge \langle e_2 \rangle = [\![e_2]\!] \wedge \langle e_1 \rangle = [\![e_1]\!] \vee \\
&[\![e_1]\!] = \texttt{TensorType}(\epsilon_1) \wedge ... \\
&\vee \\
&... \\
&\vee \\
&[\![e_1]\!] = \texttt{TensorType}(\epsilon_2) \wedge \\
&[\![e_2]\!] = \texttt{TensorType}(\sigma_1, \sigma_2) \wedge \\
&\langle e_1 \rangle = \texttt{TensorType}(\epsilon'_1, \epsilon'_2) \wedge \\
&\langle e_2 \rangle = \texttt{TensorType}(\sigma'_1, \sigma'_2) \wedge \\
&\epsilon'_1 = \sigma_1 = \sigma'_1 \wedge \\
&(\sigma_2 = \epsilon_2 = \sigma'_2 = \epsilon'_2 \vee \\
&\sigma_2 = 1 \wedge \epsilon_2 \neq 1 \wedge \sigma'_2 = \epsilon_2 = \epsilon'_2 \vee \\
&\epsilon_2 = 1 \wedge \sigma_2 \neq 1 \wedge \epsilon'_2 = \sigma_2 = \sigma'_2) \vee \\
&... \\
&\vee \\
&([\![e_1]\!] = \texttt{TensorType}(\epsilon_1, \epsilon_2, \epsilon_3, \epsilon_4) \wedge \\
&[\![e_2]\!] = \texttt{TensorType}(\sigma_1, \sigma_2, \sigma_3, \sigma_4) \wedge \\
&\langle e_1 \rangle = \texttt{TensorType}(\epsilon'_1, \epsilon'_2, \epsilon'_3, \epsilon'_4) \wedge \\
&\langle e_2 \rangle = \texttt{TensorType}(\sigma'_1, \sigma'_2, \sigma'_3, \sigma'_4) \wedge \\
&((\epsilon_1 = \sigma_1 = \epsilon'_1 = \sigma'_1) \vee \\
&((\epsilon_1 = 1 \wedge \zeta_1 \neq 1 \wedge \epsilon'_1 = \zeta_1 \wedge \zeta'_1 = \zeta_1) \vee \\
&(\zeta_1 = 1 \wedge \epsilon_1 \neq 1 \wedge \zeta'_1 = \epsilon_1 \wedge \epsilon'_1 = \epsilon_1)) \vee ... \vee \\
&(\epsilon_4 = \sigma_4 = \epsilon'_4 = \sigma'_4) \vee \\
&((\epsilon_4 = 1 \wedge \zeta_4 \neq 1 \wedge \epsilon'_4 = \zeta_4 \wedge \zeta'_4 = \zeta_4) \vee \\
&(\zeta_4 = 1 \wedge \epsilon_4 \neq 1 \wedge \zeta'_4 = \epsilon_4 \wedge \epsilon'_4 = \epsilon_4)))
\end{aligned}
$$

**Reshape constraints.**

$$From : \texttt{can-reshape}([\![e]\!], (D_1, ..., D_m))$$

$$\begin{aligned}
To : [\![e]\!] = \texttt{Dyn} \vee \\
([\![e]\!] = \texttt{TensorType}(\epsilon_1) \wedge (\epsilon_1 = \texttt{Dyn} \vee \\
\epsilon_1 \neq \texttt{Dyn} \wedge \\
\epsilon_1 = D_1 \times ... \times D_n)) \vee ... \vee \\
([\![e]\!] = \texttt{TensorType}(\epsilon_1, ..., \epsilon_4) \wedge \\
(\exists i \in \{1, ..., 5\} : \epsilon_i = \texttt{Dyn} \wedge \\
\forall \epsilon_j \neq \texttt{Dyn} : D_1 \times .... \times D_m \ mod \ \prod \epsilon_j = 0)
\end{aligned}$$

$$From : \texttt{can-reshape}(\texttt{TensorType}([\![e]\!], (D_1, ..., \texttt{Dyn}, ..., D_m))$$

$$
\begin{aligned}
To : (\llbracket e \rrbracket = \texttt{Dyn} \vee (\llbracket e \rrbracket = \texttt{TensorType}(\epsilon_1) \wedge \\
\epsilon_1 = \texttt{Dyn} \vee \\
\epsilon_1 \neq \texttt{Dyn} \wedge \\
\epsilon_1 \ mod \ D_1 \times ... \times D_m = 0) \vee ... \vee \\
(\llbracket e \rrbracket = \texttt{TensorType}(\epsilon_1, ..., \epsilon_4) \wedge \\
(\exists i \in \{1, ..., 5\} : \epsilon_i = \texttt{Dyn}) \vee \\
((\forall i \in \{1, ..., 5\} : \epsilon_i \neq \texttt{Dyn}) \wedge \\
(\prod_1^5 \epsilon_i \ mod \ D_1 \times ... \times D_m = 0 \vee D_1 \times ... \times D_m \ mod \ \prod_1^5 \epsilon_i = 0))
\end{aligned}
$$

$\sqcup^*$ **constraints.** Let $\kappa$ range over $Dyn$ and $\texttt{TensorType}(\zeta_1, ..., \zeta_m)$.

$$
From : \llbracket e \rrbracket = \langle e_1 \rangle \sqcup^* \langle e_2 \rangle
$$

$$
\begin{aligned}
((\langle e_1 \rangle = \texttt{Dyn} \vee \langle e_2 \rangle = \texttt{Dyn}) \wedge \llbracket e \rrbracket = \texttt{Dyn}) \vee \\
\forall i \in \{1, ..., 5\} (\langle e_1 \rangle = \texttt{TensorType}(\epsilon_1, ..., \epsilon_i) \wedge \\
\langle e_2 \rangle = \texttt{TensorType}(\epsilon'_1, ..., \epsilon_i) \wedge \\
\llbracket e \rrbracket = \texttt{TensorType}(\zeta_1, ...\zeta_i) \wedge \\
\zeta_1 = (\epsilon_1 \sqcup \epsilon'_1) \wedge ... \wedge \zeta_i = (\epsilon_i \sqcup \epsilon'_i))
\end{aligned}
$$

**$\sqcup$ constraints**

$$From : \epsilon = \zeta_1 \sqcup \zeta_2$$

$$To : \epsilon = \zeta_1 \wedge (\zeta_1 = \zeta_2) \vee (\epsilon = \zeta_2 \wedge (\zeta_1 = \texttt{Dyn})) \vee (\epsilon = \zeta_1 \wedge (\zeta_2 = \texttt{Dyn}))$$

**Consistency constraints.** From $D \sim \zeta$ to $\zeta = \texttt{Dyn} \vee (D = \zeta)$

From $\zeta_1 \sim \zeta_2$ to $(\zeta_1 = \zeta_2) \vee (\zeta_1 = \texttt{Dyn}) \vee (\zeta_2 = \texttt{Dyn})$

From $\langle e_1 \rangle \sim \langle e_2 \rangle$ to:

$$\begin{aligned} \langle e_1 \rangle = Dyn \vee \langle e_2 \rangle = Dyn \vee ... \vee \\ (\langle e_1 \rangle = \texttt{TensorType}(\zeta_1, ..., \zeta_4) \wedge \\ \langle e_2 \rangle = \texttt{TensorType}(\zeta'_1, ..., \zeta'_4) \wedge \\ \zeta_1 \sim \zeta'_1 \wedge ... \wedge \zeta_4 \sim \zeta'_4) \end{aligned}$$

## .5 Proof of the Order-Isomorphism

We define that $P'$ is a $\Gamma$-migration of $P$ (written $P \leq_\Gamma P'$) iff $(P \sqsubseteq P' \ \wedge \ \Gamma \vdash P' : \texttt{ok})$.

Given $P$, we define the set of $\Gamma$-migrations of $P$: $Mig_\Gamma(P) = \{P' \mid P \leq_\Gamma P'\}$.

$\forall P, \Gamma$ : if $FV(P) \subseteq Dom(\Gamma)$, then $(Mig_\Gamma(P), \sqsubseteq)$ and $(Sol(Gen(P, \Gamma)), \leq)$ are order-isomorphic.

*Proof.* If $\varphi$ is a function from type variables to types, then we define the function $G_\varphi$ from programs to programs:

$$\begin{aligned} G_\varphi(\texttt{TensorConstant}) &= \texttt{TensorConstant} \\ G_\varphi(\texttt{decl}_1, ..., \texttt{decl}_n \texttt{ return } e) &= G_\varphi(\texttt{decl}_1), ..., G_\varphi(\texttt{decl}_n) \texttt{ return } G_\varphi(e) \\ G_\varphi(x : \tau) &= x : G_\varphi(x) \\ G_\varphi(e) &= e \end{aligned}$$

Let $P, \Gamma$ be given; they remain fixed in the remainder of the proof. Now we define the following function $\alpha_P$ with the help of $G_\varphi$:

$$\begin{aligned} \alpha_P &: Sol(Gen(P, \Gamma)) \to Mig_\Gamma(P) \\ \alpha_P(\varphi) &= G_\varphi(P) \end{aligned}$$

Notice that $\Gamma$ plays no role in the definitions of $G_\varphi$ and $\alpha_e$. We will show that $\alpha_P$ is a well-defined order-isomorphism. We will do this in four steps: we will show that $\alpha_P$ is well defined, injective, and surjective, and that it preserves order.

**Well defined.** We will show that if $\varphi \in Sol(Gen(P, \Gamma))$, then $\alpha_P(\varphi) \in Mig_\Gamma(P)$. Suppose $\varphi \in Sol(Gen(P, \Gamma))$. We must show

$$P \sqsubseteq \alpha_P(\varphi) \text{ and } \Gamma \vdash \alpha_P(\varphi) : \texttt{ok}.$$

In order to show $P \sqsubseteq \alpha_P(\varphi)$, notice that $P$ and $\alpha_P(\varphi)$ differ only in the type annotations of bound variables. If we have no bound variables in $P$, then $P = \alpha_P(\varphi)$. Otherwise, notice

that for every declaration of $x : \tau$ in $P$, we have that $\varphi \models \tau \sqsubseteq x$ and $G_\varphi(x : \tau) = x : \varphi(\tau)$. So we know that $P \sqsubseteq \alpha_P(\varphi)$.

Define $\mathit{Extend}(\Gamma, P)$ to be $\Gamma$ extended with $(x : \tau)$ for each declaration $x : \tau$ in $P$. In order to show $\Gamma \vdash \alpha_P(\varphi) : \texttt{ok}$, we first show that:

For a return expression $E$ in $P$, $\forall E'$ subterm of $E : \mathit{Extend}(\Gamma, G_\varphi(P)) \vdash G_\varphi(E') : \varphi([\![E']\!])$.

and then we get that $\Gamma \vdash P : \texttt{ok}$ from *ok-prog*.

We proceed by induction on $E'$.

Case: $e' = x$, where $x$ is free in $E$. Notice that $\varphi \models [\![e']\!] = \Gamma(x)$ and $\mathit{Extend}(\Gamma, G_\varphi(P))(x) = \Gamma(x)$ and use *t-var*.

Case: $e' = x$, where $x$ is bound in $e$. Notice that $\varphi \models [\![e']\!] = x$ and $\mathit{Extend}(\Gamma, G_\varphi(P))(x) = \varphi(x)$ and use *t-var*.

Case: $e' = \texttt{add}(e_1, e_2)$. Notice that $\varphi \models [\![e_1]\!] = \langle e_1 \rangle \sqcup^* \langle e_2 \rangle$ and
$\varphi \models (\langle e_1 \rangle, \langle e_2 \rangle) = \texttt{apply-broadcasting}([\![e_1]\!], [\![e_2]\!])$ and $\varphi \models \langle e_1 \rangle \sim \langle e_2 \rangle$ From the induction hypothesis we have $\mathit{Extend}(\Gamma, G_\varphi(e_1)) \vdash G_\varphi(e_1) : \varphi([\![e_1]\!])$ and $\mathit{Extend}(\Gamma, G_\varphi(e_2)) \vdash G_\varphi(e_2) : \varphi([\![e_2]\!])$. Now we use *T-Add*

Case: $e' = \texttt{reshape}(e_0, \delta)$.
We have that $\varphi \models [\![\texttt{reshape}(e_0, \delta)]\!] = \delta$ and $\varphi \models \texttt{can-reshape}([\![e_0]\!], \delta)$. By induction, we have $\mathit{Extend}(\Gamma, G_\varphi(P)) \vdash G_\varphi(e_0) : \varphi([\![e_0]\!])$. Consider the definition of $\varphi \models \texttt{can-reshape}([\![e_0]\!], \delta)$. We have that if `Dyn` *does not occur in* $\delta$ *and* $\varphi([\![e_0]\!]) \prod \delta = \prod \varphi([\![e_0]\!])$ then we can use *t-reshape-s*. Otherwise, based on the occurrences of `Dyn` in $\varphi([\![e_0]\!])$ and $\delta$, we can use *t-reshape-g* or *t-reshape*.

Case: $\texttt{Conv2D}(c_{in}, c_{out}, \kappa_{kernel}, e_0)$. We have $\varphi \models [\![e_0]\!] \rhd \texttt{TensorType}(\zeta_1, \zeta_2, \zeta_3, \zeta_4)$ and $\varphi \models c_{in} \sim \zeta_2$ and $\varphi \models [\![\texttt{Conv2D}(c_{in}, c_{out}, \kappa_{kernel}, e_0)]\!] = \texttt{calc-conv}([\![e_0]\!], c_{out}, \kappa_{kernel})$. By induction, we get that $\mathit{Extend}(\Gamma, G_\varphi(P)) \vdash G_\varphi(e) : \varphi([\![e_0]\!])$. Then we use *t-conv*.

**Injective.** We will show that $\alpha_P$ is injective, that is, we will show that

$$\text{if } \alpha_P(\varphi) = \alpha_P(\varphi'), \text{ then } \varphi = \varphi'.$$

Suppose $\alpha_P(\varphi) = \alpha_P(\varphi')$.

From the definition of $\alpha_P$ we see that for every declaration $x : \tau$ in $P$ we have $\varphi(x) = \varphi'(x)$. We will show that for every declaration $x : \tau$, $\varphi(x) = \varphi'([\![x]\!])$. Note that for every variable declaration $x : \tau$, we have that $\varphi \models \tau \sqsubseteq x$ and $\varphi' \models \tau \sqsubseteq x$ and since $\alpha_P(\varphi) = \alpha_P(\varphi')$ then $\varphi(x) = \varphi'(x)$.

Next we show that for every occurrence of a subterm $e'$ in the return expression $e$, we have $\varphi([\![e']\!]) = \varphi'([\![e']\!])$, and for every occurrence of a subterm $\texttt{add}(e_1, e_2)$, we have that $\varphi(\langle e_1 \rangle) = \varphi(\langle e_1' \rangle)$ and $\varphi(\langle e_1 \rangle) = \varphi(\langle e_1' \rangle)$. We proceed by induction on $E'$.

Case: $e' = \texttt{TensorConstant}$. From $\varphi \models [\![e']\!] = \mathit{shape}(\texttt{TensorConstant})$ and $\varphi' \models [\![e']\!] = \mathit{shape}(\texttt{TensorConstant})$ , we have $\varphi([\![e']\!]) = \mathit{shape}(\texttt{TensorConstant}) = \varphi'([\![e']\!])$.

Case: $e' = x$, where $x$ is bound in $E$. From $\varphi \models [\![e']\!] = x$ and $\varphi' \models [\![e']\!] = x$, we have $\varphi([\![e']\!]) = \varphi(x) = \varphi'(x) = \varphi'([\![e']\!])$.

Case: $e' = \texttt{reshape}(e_0, \delta)$. From the induction hypothesis, we have $\varphi([\![e_0]\!]) = \varphi'([\![e_0]\!])$. From $\varphi \models \texttt{can-reshape}([\![e_0]\!], \delta)$ and $\varphi' \models \texttt{can-reshape}([\![e_0]\!], \delta)$ we have $\varphi([\![e']\!]) = \varphi'([\![e']\!]) = \delta$.

Case $e' = \texttt{Conv2D}(c_{in}, c_{out}, \kappa_{kernel}, e_0)$. From the induction hypothesis, we have $\varphi([\![e_0]\!]) = \varphi'([\![e_0]\!])$. From $\varphi \models [\![e_0]\!] \rhd \texttt{TensorType}(\zeta_1, \zeta_2, \zeta_3, \zeta_4)$ and $\varphi' \models [\![e_0]\!] \rhd \texttt{TensorType}(\zeta_1, \zeta_2, \zeta_3, \zeta_4)$,

$\varphi \models c_{in} \sim \zeta_2$ and
$\varphi' \models c_{in} \sim \zeta_2$ and $\varphi \models [\![\texttt{Conv2D}(c_{in}, c_{out}, \kappa_{kernel}, e_0)]\!] = \texttt{calc-conv}([\![e_0]\!], c_{out}, \kappa_{kernel})$ and $\varphi' \models [\![\texttt{Conv2D}(c_{in}, c_{out}, \kappa_{kernel}, e_0)]\!] = \texttt{calc-conv}([\![e_0]\!], c_{out}, \kappa_{kernel})$ we have that $\varphi([\![e']\!]) = \varphi'([\![e']\!])$.

Case $e' = \texttt{add}(e_1, e_2)$. From the induction hypothesis, we have $\varphi([\![e_1]\!]) = \varphi'([\![e_1]\!])$ and $\varphi([\![e_2]\!]) = \varphi'([\![e_2]\!])$. Then we have $\varphi \models (\langle e_1 \rangle, \langle e_2 \rangle) = \texttt{apply-broadcasting}([\![e_1]\!], [\![e_2]\!])$ and $\varphi' \models (\langle e_1 \rangle, \langle e_2 \rangle) = \texttt{apply-broadcasting}([\![e_1]\!], [\![e_2]\!])$ and $\varphi \models \langle e_1 \rangle \sim \langle e_2 \rangle$ and $\varphi' \models (\langle e_1 \rangle, \langle e_2 \rangle) = \texttt{apply-broadcasting}([\![e_1]\!], [\![e_2]\!])$. So we have that $\varphi([\![e']\!]) = \varphi'([\![e']\!])$.

**Surjective.** We will show that $\alpha_P$ is surjective, that is, we will show that

$$\text{if } P_0 \in \mathit{Mig}_\Gamma(P), \text{ then } \exists \varphi \in \mathit{Sol}(\mathit{Gen}(P, \Gamma)) : P_0 = \alpha_P(\varphi).$$

From $P_0 \in \mathit{Mig}_\Gamma(P)$ we have $P \sqsubseteq P_0$ and $\Gamma \vdash P_0 : \texttt{ok}$. From $\Gamma \vdash P_0 : \texttt{ok}$ we have that $\mathit{Extend}(\Gamma, P_0) \vdash P_0 : \texttt{ok}$.

We define $\varphi$ as follows. Consider a derivation $D$ of $\mathit{Extend}(\Gamma, P_0) \vdash P_0 : \texttt{ok}$. First, for $x \in \mathit{Dom}(\mathit{Extend}(\Gamma, P_0))$, define $\varphi(x) = \mathit{Extend}(\Gamma, P_0)(x)$. Second, for every occurrence of a subterm $e'$ of the return expression $e_0$, find the judgment in $D$ of the form $\Gamma' \vdash e' : \tau'$, and define $\varphi([\![e']\!]) = \tau'$. Then for the subterm $e'$ of the form $\texttt{add}(e_1, e_2)$ in $e_0$, find the use of *T-Add* for $e'$ and in that use, find the equation $((\tau_1, \tau_2) = \texttt{apply-broadcasting}(t_1, t_2)$ , and define $\varphi(\langle e_1 \rangle) = \tau_1$ and $\varphi(\langle e_2 \rangle) = \tau_2$.

We must show that $\varphi \in \mathit{Sol}(\mathit{Gen}(P, \Gamma))$. First note that for every variable declaration $x : \tau$ we have that $\varphi(x) = \tau$.

Next, we will do a case analysis of the occurrences of subterms $e'$ in the return expression $e$.

Case: $e' = \texttt{TensorConstant}$. We have that $\varphi([\![e']\!]) = \mathit{shape}(\texttt{TensorConstant})$ so $\varphi \models [\![e']\!] = \mathit{shape}(\texttt{TensorConstant})$.

Case: $e' = x$, where $x$ is bound in $E$. From *(t-var)* we have that $\varphi([\![e']\!]) = \varphi(x)$ so $\varphi \models [\![e']\!] = x$.

Case: $e' = \texttt{add}(e_1, e_2) : \tau_1$. The derivation $D$ contains this use of *T-Add*:

$$\frac{\Gamma \vdash e_1 : t_1 \;\; \Gamma \vdash e_1 : t_2 \;\; (\tau_2, \tau_2) = \texttt{apply-broadcasting}(t_1, t_2) \;\; \tau_1 \sim \tau_2}{\Gamma \vdash \texttt{add}(e_1, e_2) : \tau_1 \sqcup^* \tau_2} \; \textit{(t-add)}$$

So, $\varphi([\![e_1]\!]) = \tau_1$ and $\varphi([\![e_2]\!]) = \tau_2$. By examining our constraints and the fact that $\alpha_P(\varphi) = G_\varphi(P) = P_0$, we are done. We know that $\alpha_P(\varphi) = G_\varphi(P) = P_0$ is that $P_0$ differs from $P$ only in the type annotations of variable declarations.

Case $e' = \texttt{Conv2D}(c_{in}, c_{out}, \kappa_{kernel}, e)$. We consider the use of *T-Conv2D* and inspect the constraints and apply the reasoning above.

Case $e' = \texttt{reshape}(e', \delta)$. We consider the use of either *T-reshape-s*, *T-reshape* or *T-reshape-g* and inspect the constraints and apply the reasoning above.

**Preserves order.** We will show that $\alpha_P$ preserves order, that is, we will show that

$$\text{if } \varphi \leq \varphi', \text{ then } \alpha_P(\varphi) \sqsubseteq \alpha_P(\varphi').$$

Suppose that $\varphi \leq \varphi'$.

First, for variable declarations $x : \tau$, from $\varphi \leq \varphi'$ we have $\varphi(x) \sqsubseteq \varphi'(x)$. From the definition of $G_\varphi$ and *p-decl*, we have that $G_\varphi(x : \tau) = x : \varphi(x) \sqsubseteq x : \varphi'(x) = G_{\varphi'}(x : \tau)$

For the return expression $e$, we want to show the stronger statement that

$$\text{if } \varphi \leq \varphi', \text{ then } \forall e' : G_{\varphi}(e') = G_{\varphi'}(e').$$

Note that this is straightforward. The reason is that $G_{\varphi'}(e') = e' = G_{\varphi'}(e')$

□

## REFERENCES


[AAB16] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mane, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viegas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. "TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems.", 2016.

[AF19] Pedro Ângelo and Mário Florido. "Type Inference for Rank 2 Gradual Intersection Types." In *TFP*, 2019.

[Ans22] Jason Ansel. "TorchDynamo.", 1 2022.

[BAT14] Gavin Bierman, Martín Abadi, and Mads Torgersen. "Understanding TypeScript." In Richard Jones, editor, *ECOOP 2014 – Object-Oriented Programming*, pp. 257–281, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[BCD83] H. P. Barendregt, M. Coppo, and M. Dezani-Ciancaglini. "A Filter Lambda Model and the Completeness of Type Assignment." *Journal of Symbolic Logic*, **48**:931–940, 1983.

[BDD95] Franco Barbanera, Mariangiola Dezani-Ciancaglini, and Ugo De'Liguoro. "Intersection and Union Types: Syntax and Semantics." *Information and Computation*, **119**(2):202–230, 1995.

[BFH18] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. "JAX: composable transformations of Python+NumPy programs.", 2018.

[CCE18] John Peter Campora, Sheng Chen, Martin Erwig, and Eric Walkingshaw. "Migrating Gradual Types." *Proc. ACM Program. Lang.*, **2**(POPL):15:1–15:29, 2018.

[CF91] Robert Cartwright and Mike Fagan. "Soft Typing." In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementa-*

*tion*, PLDI '91, p. 278–292, New York, NY, USA, 1991. Association for Computing Machinery.

[CL17] Giuseppe Castagna and Victor Lanvin. "Gradual Typing with Union and Intersection Types." **1**(ICFP), August 2017.

[CLP19] Giuseppe Castagna, Victor Lanvin, Tommaso Petrucciani, and Jeremy G. Siek. "Gradual Typing: A New Perspective." *Proc. ACM Program. Lang.*, **3**(POPL):16:1–16:32, January 2019.

[CS16] Matteo Cimini and Jeremy Siek. "The Gradualizer: A Methodology and Algorithm for Generating Gradual Type Systems." In *Proceedings of POPL'16, ACM Symposium on Principles of Programming Languages*, New York, 2016. ACM.

[CT21] Fernando Cristiani and Peter Thiemann. "Generation of TypeScript Declaration Files from JavaScript Code." In *International Conference on Managed Programming Languages and Runtimes*, pp. 97–112, 2021.

[DP00] Rowan Davies and Frank Pfenning. "Intersection Types and Computational Effects." In *ICFP*, pp. 198–208, 2000.

[DP03] Jana Dunfield and Frank Pfenning. "Type Assignment for Intersections and Unions in Call-by-Value Languages." In *FoSSaCS*, pp. 250–266, 2003.

[Dun14] Jana Dunfield. "Elaborating Intersection and Union Types." *Journal of Functional Programming*, **24**(2–3):133–165, 2014.

[FFK96] Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, Stephanie Weirich, and Matthias Felleisen. "Catching Bugs in the Web of Program Invariants." In *Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation*, PLDI '96, p. 23–32, New York, NY, USA, 1996. Association for Computing Machinery.

[FM14] Asger Feldthaus and Anders Møller. "Checking Correctness of TypeScript Interfaces for JavaScript Libraries." pp. 1–16, 2014.

[FP91] Tim Freeman and Frank Pfenning. "Refinement Types for ML." In *PLDI*, pp. 268–277, 1991.

[GC15] Ronald Garcia and Matteo Cimini. "Principal Type Schemes for Gradual Programs." In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 303–315, 2015.

[GCT16] Ronald Garcia, Alison M. Clark, and Éric Tanter. "Abstracting Gradual Typing." In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 429–442, New York, 2016. ACM.

[GF18] Ben Greenman and Matthias Felleisen. "A Spectrum of Type Soundness and Performance." *Proc. ACM Program. Lang.*, **2**(ICFP), jul 2018.

[GM18] Ben Greenman and Zeina Migeed. "On the cost of Type-Tag Soundness." In *PEPM*, 2018.

[GR88] Paola Giannini and Simona Ronchi Della Rocca. "Characterization of Typings in Polymorphic Type Discipline." In *Proceedings of LICS'88, Third Annual Symposium on Logic in Computer Science*, pp. 61–70, 1988.

[GR13] Jacques Garrigue and Didier Rémy. "Ambivalent Types for Principal Type Inference with GADTs." In Chung-chieh Shan, editor, *Programming Languages and Systems - 11th Asian Symposium, APLAS 2013, Melbourne, VIC, Australia, December 9-11, 2013. Proceedings*, volume 8301 of *Lecture Notes in Computer Science*, pp. 257–272. Springer, 2013.

[Hen93] Fritz Henglein. "Type Inference with Polymorphic Recursion." *ACM Trans. Program. Lang. Syst.*, **15**(2):253–289, apr 1993.

[HFS22] Joshua Hoeflich, Robert Bruce Findler, and Manuel Serrano. "Highly Illogical, Kirk: Spotting Type Mismatches in the Large despite Broken Contracts, Unsound Types, and Too Many Linters." **6**(OOPSLA2), 2022.

[Hin82] J. Roger Hindley. "Principal type-schemes for functional programs." *Journal of the ACM (JACM)*, **29**(2):233–243, 4 1982.

[HKS22] Momoko Hattori, Naoki Kobayashi, and Ryosuke Sato. "Gradual Tensor Shape Checking.", 2022.

[HMS16] Thomas S. Heinze, Anders Møller, and Fabio Strocco. "Type Safety Analysis for Dart." In *DLS*, 2016.

[HUE18] Mostafa Hassan, Caterina Urban, Marco Eilers, and Peter Müller. "MaxSMT-Based Type Inference for Python 3." In *Proceedings of CAV'18, Computer-Aided Verification*, 2018.

[Jav22] Several JavaDocumenters. "Interface IntersectionType." https://docs.oracle.com/en/java/javase/14/docs/api/java.compiler/javax/lang/model/type/IntersectionType.html, Accessed Oct 19 2022.

[Jim95] Trevor Jim. "Rank 2 Type Systems and Recursive Definitions." Technical Report LCS/TM–531, MIT, 1995.

[JKS22] Ho Young Jhoo, Sehoon Kim, Woosung Song, Kyuyeon Park, DongKwon Lee, and Kwangkeun Yi. "A Static Analyzer for Detecting Tensor Shape Errors in Deep Neural Network Training Code." In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*, ICSE '22, p. 337–338, New York, NY, USA, 2022. Association for Computing Machinery.

[JMT09] Simon Holm Jensen, Anders Møller, and Peter Thiemann. "Type Analysis for JavaScript." In *Proc. 16th International Static Analysis Symposium (SAS)*, volume 5673 of *LNCS*. Springer-Verlag, August 2009.

[KKT15] Andrew M. Kent, David Kempe, and Sam Tobin-Hochstadt. "Occurrence Typing Modulo Theories." *CoRR*, **abs/1511.07033**, 2015.

[KM17] Erik Krogh Kristensen and Anders Møller. "Type Test Scripts for TypeScript Testing." **1**, 2017.

[KW04] A.J. Kfoury and J.B. Wells. "Principality and type inference for intersection types using expansion variables." *Theoretical Computer Science*, **311**(1):1–70, 2004.

[LDG20] Sifis Lagouvardos, Julian T Dolby, Neville Grech, Anastasios Antoniadis, and Yannis Smaragdakis. "Static Analysis of Shape in TensorFlow Programs." In *ECOOP*, Germany, 2020. LIPICS.

[Lei83] Daniel Leivant. "Polymorphic Type Inference." In *Proceedings of the 10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '83, p. 88–98, New York, NY, USA, 1983. Association for Computing Machinery.

[LGF21] Lukas Lazarek, Ben Greenman, Matthias Felleisen, and Christos Dimoulas. "How to Evaluate Blame for Gradual Types." *Proc. ACM Program. Lang.*, **5**(ICFP), aug 2021.

[LT17] Nico Lehmann and Éric Tanter. "Gradual Refinement Types." In *Proceedings of POPL, ACM Symposium on Principles of Programming Languages*, 2017.

[Mil78] Robin Milner. "A Theory of Type Polymorphism in Programming." *Journal of Computer and System Sciences*, **17**:348–375, 1978.

[Mog92] Torben Æ. Mogensen. "Efficient Self-Interpretations in Lambda Calculus." *Journal of Functional Programming*, **2**(3):345–363, 1992. See also DIKU Report D–128, Sep 2, 1994.

[MP19] Zeina Migeed and Jens Palsberg. "What is Decidable about Gradual Types?" *Proc. ACM Program. Lang.*, **4**(POPL), December 2019.

[MSI18] Yusuke Miyazaki, Taro Sekiyama, and Atsushi Igarashi. "Dynamic Type Inference for Gradual Hindley-Milner Typing." *CoRR*, **abs/1810.12619**, 2018.

[NM04] Peter Møller Neergaard and Harry G. Mairson. "Types, Potency, and Idempotency: Why Nonlinearity and Amnesia Make a Type System Work." *SIGPLAN Not.*, **39**(9):138–149, sep 2004.

[OSA16] Bruno C. d. S. Oliveira, Zhiyuan Shi, and João Alpuim. "Disjoint Intersection Types." In *ICFP*, pp. 364–377, 2016.

[PAG21] Luna Phipps-Costin, Carolyn Jane Anderson, Michael Greenberg, and Arjun Guha. "Solver-Based Gradual Type Migration." In *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA)*, 2021.

[PGC17] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. "Automatic Differentiation in PyTorch." In *NIPS 2017 Workshop on Autodiff*, 2017.

[PGM19] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. "PyTorch: An Imperative Style, High-Performance Deep Learning Library." In *Advances in Neural Information Processing Systems 32*, pp. 8024–8035. Curran Associates, Inc., 2019.

[PS91] Jens Palsberg and Michael I. Schwartzbach. "Object-Oriented Type Inference." *SIGPLAN Not.*, **26**(11):146–161, nov 1991.

[PS21] Adam Paszke and Brennan Saeta. "Tensors Fitting Perfectly.", 2021.

[PT00] Benjamin C. Pierce and David N. Turner. "Local Type Inference." *ACM Trans. Program. Lang. Syst.*, **22**(1):1–44, jan 2000.

[RCH12] Aseem Rastogi, Avik Chaudhuri, and Basil Hosmer. "The Ins and Outs of Gradual Type Inference." *SIGPLAN Not.*, **47**(1):481–494, 2012.

[RDH21] James K. Reed, Zachary DeVito, Horace He, Ansley Ussery, and Jason Ansel. "Torch.fx: Practical Program Capture and Transformation for Deep Learning in Python.", 2021.

[Rem05] Didier Rémy. "Simple, partial type-inference for System F based on type-containment." In Olivier Danvy and Benjamin C. Pierce, editors, *Proceedings of the 10th ACM SIGPLAN International Conference on Functional Programming, ICFP 2005, Tallinn, Estonia, September 26-28, 2005*, pp. 130–143. ACM, 2005.

[Rey88] John C. Reynolds. "Preliminary Design of the programming language Forsythe." Technical Report CMU-CS-88-159, Carnegie Mellon University, 1988. http://doi.library.cmu.edu/10.1184/OCLC/18612825.

[Rey96] John C. Reynolds. "Design of the programming language Forsythe." Technical Report CMU-CS-96-146, Carnegie Mellon University, 1996.

[Rin18] Norman A. Rink. "Modeling of languages for tensor manipulation." *ArXiv*, **abs/1801.08771**, 2018.

[RLK19] Jared Roesch, Steven Lyubomirsky, Marisa Kirisame, Josh Pollock, Logan Weber, Ziheng Jiang, Tianqi Chen, Thierry Moreau, and Zachary Tatlock. "Relay: A High-Level IR for Deep Learning." *CoRR*, **abs/1904.08368**, 2019.

[RLW18] Jared Roesch, Steven Lyubomirsky, Logan Weber, Josh Pollock, Marisa Kirisame, Tianqi Chen, and Zachary Tatlock. "Relay: a new IR for machine learning frameworks." In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*. ACM, jun 2018.

[RY08] Didier Rémy and Boris Yakobowski. "From ML to ML

F

: graphic type constraints with efficient type inference." In James Hook and Peter Thiemann, editors, *Proceeding of the 13th ACM SIGPLAN international conference on Functional programming, ICFP 2008, Victoria, BC, Canada, September 20-28, 2008*, pp. 63–74. ACM, 2008.

[SB05] Sanjit A. Seshia and Randal E. Bryant. "Deciding Quantifier-Free Presburger Formulas Using Parameterized Solution Bounds." *Logical Methods in Computer Science*, **1**:1–26, 2005.

[Sca22] Several ScalaDocumenters. "Intersection Types." https://dotty.epfl.ch/docs/reference/new-types/intersection-types.html, Accessed Oct 19 2022.

[Sha22] Tushar Sharma. "Demystifying Typescript Unions and Intersections." https://www.serverlessguru.com/blog/demystifying-typescript-unions-and-intersection, 2022.

[SLZ18] Chengnian Sun, Yuanbo Li, Qirun Zhang, Tianxiao Gu, and Zhendong Su. "Perses: Syntax-Guided Program Reduction." In *ICSE'18, International Conference on Software Engineering*, 2018.

[SMS18] Justin Slepak, Panagiotis Manolios, and Olin Shivers. "Rank Polymorphism Viewed as a Constraint Problem." In *Proceedings of the 5th ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*, ARRAY 2018, p. 34–41, New York, NY, USA, 2018. Association for Computing Machinery.

[SSM19] Justin Slepak, Olin Shivers, and Panagiotis Manolios. "Records with Rank Polymorphism." In *Proceedings of the 6th ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming*, ARRAY 2019, p. 80–92, New York, NY, USA, 2019. Association for Computing Machinery.

[ST06] Jeremy G. Siek and Walid Taha. "Gradual Typing for Functional Languages." In *IN SCHEME AND FUNCTIONAL PROGRAMMING WORKSHOP*, pp. 81–92, 2006.

[ST07] Jeremy Siek and Walid Taha. "Gradual Typing for Objects." In *Proceedings of the 21st European Conference on Object-Oriented Programming*, pp. 2–27, 2007.

[ST13] Vincent St-Amour and Neil Toronto. "Experience Report: Applying Random Testing to a Base Type Environment." pp. 351–356, 2013.

[SV08] Jeremy G. Siek and Manish Vachharajani. "Gradual Typing with Unification-based Inference." In *Proceedings of the 2008 Symposium on Dynamic Languages*, pp. 7:1–7:12, 2008.

[SVC15a] Jeremy G. Siek, Michael M. Vitousek, Matteo Cimini, and John Tang Boyland. "Refined Criteria for Gradual Typing." In *SNAPL*, pp. 274–293, Germany, 2015. LIPICS.

[SVC15b] Jeremy G. Siek, Michael M. Vitousek, Matteo Cimini, Sam Tobin-Hochstadt, and Ronald Garcia. "Monotonic References for Efficient Gradual Typing." In

*Proceedings of the 24th European Symposium on Programming on Programming Languages and Systems - Volume 9032*, pp. 432–456, 2015.

[TF08] Sam Tobin-Hochstadt and Matthias Felleisen. "The Design and Implementation of Typed Scheme." In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '08, pp. 395–406, New York, NY, USA, 2008. ACM.

[TFF17] Sam Tobin-Hochstadt, Matthias Felleisen, Robert Findler, Matthew Flatt, Ben Greenman, Andrew M. Kent, Vincent St-Amour, T. Stephen Strickland, and Asumu Takikawa. "Migratory Typing: Ten Years Later." In Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi, editors, *2nd Summit on Advances in Programming Languages (SNAPL 2017)*, volume 71 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pp. 17:1–17:17, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

*[Keywords: design principles, type systems, gradual typing.]*

[TFG16] Asumu Takikawa, Daniel Feltey, Ben Greenman, Max S. New, Jan Vitek, and Matthias Felleisen. "Is Sound Gradual Typing Dead?" In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 456–468, 2016.

[Typ22] Several TypeScriptDocumenters. "TypeScript 1.6." https://www.typescriptlang.org/docs/handbook/release-notes/typescript-1-6.html, Accessed Oct 19 2022.

[VKS14] Michael M. Vitousek, Andrew M. Kent, Jeremy G. Siek, and Jim Baker. "Design and Evaluation of Gradual Typing for Python." *SIGPLAN Not.*, **50**(2):45–56, 2014.

[VS20] Sahil Verma and Zhendong Su. "ShapeFlow: Dynamic Shape Interpreter for TensorFlow.", 2020.

[VSS17] Michael M. Vitousek, Cameron Swords, and Jeremy G. Siek. "Big Types in Little Runtime: Open-world Soundness and Collaborative Blame for Gradual Type Systems." *SIGPLAN Not.*, **52**(1):762–774, 2017.

[Wam20] Dean Wampler. "Scala 3: Intersection and Union Types." https://medium.com/scala-3/intersection-and-union-types-860665b785c1, 2020.

[WC97] Andrew K. Wright and Robert Cartwright. "A Practical Soft Type System for Scheme." *ACM Trans. Program. Lang. Syst.*, **19**(1):87–152, January 1997.

[WDS20] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Perric Cistac, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. "Transformers: State-of-the-Art Natural Language Processing." pp. 38–45. Association for Computational Linguistics, 10 2020.

[WMW17] Jack Williams, J. Garrett Morris, Philip Wadler, and Jakub Zalewski. "Mixed Messages: Measuring Conformance and Non-Interference in TypeScript." pp. 28:1–28:29, 2017.

[Wol00] Wayne Wolf. *Computers as Components, Principles of Embedded Computing System Design.* Morgan Kaufman Publishers, 2000.

[XZC16] Zhaogui Xu, Xiangyu Zhang, Lin Chen, Kexin Pei, and Baowen Xu. "Python probabilistic type inference with natural language support." In *Proceedings of 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, pp. 607–618, 2016.

[Zar19] Saeed Zarinfam. "Java Generics: Intersection Types." https://itnext.io/java-generics-intersection-types-23b2fbdddfbb, 2019.

[ZCC18] Yuhao Zhang, Yifan Chen, Shing-Chi Cheung, Yingfei Xiong, and Lu Zhang. "An Empirical Study on TensorFlow Program Bugs." In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2018, p. 129–140, New York, NY, USA, 2018. Association for Computing Machinery.