What Is Decidable about Gradual Types?

ZEINA MIGEED and JENS PALSBERG, University of California, Los Angeles (UCLA), USA

Programmers can use gradual types to migrate programs to have more precise type annotations and thereby improve their readability, efficiency, and safety. Such migration requires an exploration of the migration space and can benefit from tool support, as shown in previous work. 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.

CCS Concepts: • **Theory of computation → Type structures**.

Additional Key Words and Phrases: Types, migration, algorithms

ACM Reference Format:
Zeina Migeed and Jens Palsberg. 2020. What Is Decidable about Gradual Types?. *Proc. ACM Program. Lang.* 4, POPL, Article 29 (January 2020), 29 pages. https://doi.org/10.1145/3371097

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 paper we will focus on one such combination, namely the well-known *gradual typing* of Siek and Taha [2006]. 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 [Siek et al. 2015a].

Gradual typing has found practical application in Typed Racket [Tobin-Hochstadt and Felleisen 2008], TypeScript [Bierman et al. 2014], Reticulated Python [Vitousek et al. 2014], 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 Siek and Taha [2006] 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,

Authors' address: Zeina Migeed, zeina@cs.ucla.edu; Jens Palsberg, palsberg@ucla.edu, Computer Science Department, University of California, Los Angeles (UCLA), 4732 Boelter Hall, Los Angeles, CA, 90095, USA.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
© 2020 Copyright held by the owner/author(s).
2475-1421/2020/1-ART29
https://doi.org/10.1145/3371097

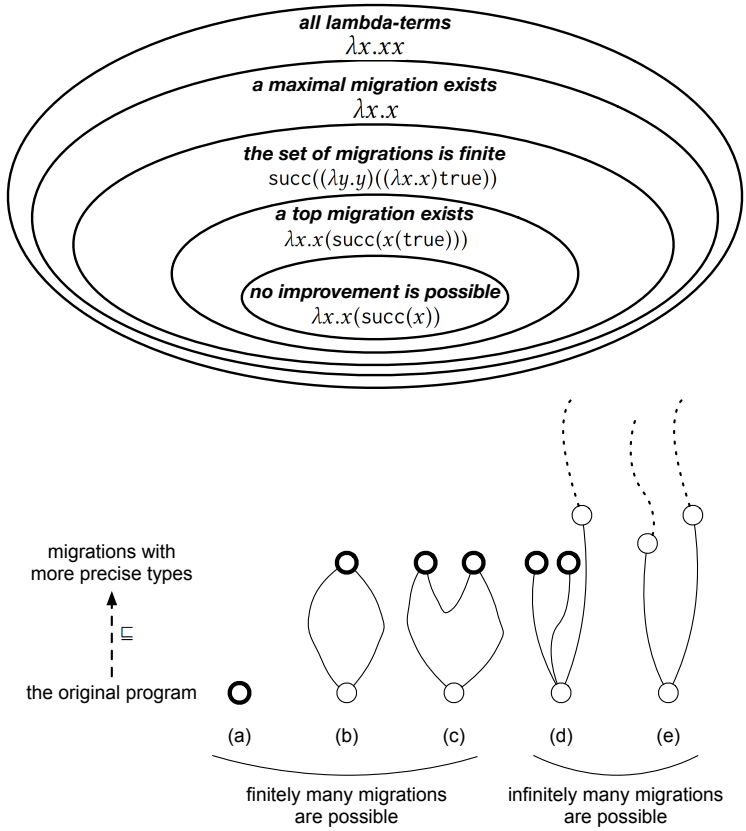

Fig. 1. Nested sets of λ-terms: (a) no improvement is possible; (b) a top migration exists; (c) the set of migrations is finite; (d) a maximal migration exists; (e) all lambda-terms.

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.

Motivation. When we face a migration task, 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. However, 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: *is any improvement possible?* (Singleton problem), *does a top migration exist?* (Top-choice problem), *is the set of migrations finite?* (Finiteness problem), and *does a maximal migration exist?* (Maximality problem). Information about which kind of program we are facing will help us figure out how long we should continue a migration exploration.

We can use those properties to classify programs into five increasingly larger sets. Figure 1 illustrates both how those sets are nested (top half) and the migration possibilities (bottom half). Each of (a)–(e) shows the original program as a circle and, above it, possible migrations; maximal migrations are shown as bold circles.

Our contributions. We present algorithms (with names in bold below) and a hardness result for deciding the four questions above for the gradually typed λ-calculus [Siek and Taha 2006].

Singleton problem:	decidable in $O(n^2)$ time (Theorem 3.5)	**Singleton Checker**
Top-choice problem:	decidable in EXPTIME (Theorem 5.3)	**Top-Choice Checker**
Finiteness problem:	decidable in EXPTIME (Theorem 4.11)	**Finiteness Checker**
Maximality problem:	NP-hard (Theorem 6.1).	

In Section 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) relies on a theorem known as *the static gradual guarantee* [Siek et al. 2015a] 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 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 5) first runs our finiteness checker and then searches the set of migrations. Our NP-hardness proof (Section 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 7). We discuss related work in Section 8. Briefly, the most closely related work is the POPL 2018 paper by Campora, Chen, Erwig, and Walkingshaw Campora et al. [2018], 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 Campora et al. [2018] may produce non-maximal migrations (see Sections 7–8), which makes it unsuitable for our decision problems.

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

2 THE GRADUALLY TYPED LAMBDA CALCULUS

2.1 Syntax and Type System

Figure 2 shows the gradually typed λ-calculus [Siek and Taha 2006], in the convenient reformulation by Cimini and Siek [2016]. 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 type rules for Booleans (*T-True* and *T-False*) and numbers (*T-Num*) are straightforward, and the type rules for variables (*T-Var*) and abstractions (*T-Abs*) are as in simply-typed λ-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 λ-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

Syntax:

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

Type rules:

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

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

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

Consistency:

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

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

Matching:

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

Precision:

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

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

Fig. 2. The gradually typed λ-calculus.

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 1.

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

Theorem 2.1 (Unique Type). $\forall E, \Gamma, T, T'$, *if* $\Gamma \vdash E : T$ *and* $\Gamma \vdash E : T'$, *then* $T = T'$.

Theorem 2.2 (Weakening). $\forall E, \Gamma, T, T' :$ *if* $x \notin FV(E)$, *then* $\Gamma \vdash E : T$ *iff* $\Gamma, x : T' \vdash E : T$.

Theorem 2.3 (Static Gradual Guarantee [Siek et al. 2015a]). $\forall E, E', \Gamma, T : \exists T' :$ *if* $\Gamma \vdash E : T \ \wedge\ E' \sqsubseteq E$ *then* $\Gamma \vdash E' : T' \ \wedge\ T' \sqsubseteq T$.

Theorem 2.4 (Finite Intervals). $\forall E_l, E_u : \{\ E \mid E_l \sqsubseteq E \sqsubseteq E_u\ \}$ *is finite.*

The type system assigns at most one type to every program, as expressed by Theorem 2.1. The reason is that every bound variable is declared with a type. Given E, Γ, 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 2.2 is a standard result about adding and removing parts of the environment that is true of many type systems. The static gradual guarantee (Theorem 2.3) 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 2.4).

In Appendix A of the supplementary material we prove Theorem 2.1. We omit the standard proof of Theorem 2.2. Siek et al. [2015a] proved Theorem 2.3. The proof of Theorem 2.4 is straightforward and omitted.

2.2 Decision Problems

We define that E' is a Γ-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 Γ-migration of E if E' improves E and E' type checks.

Given E, we define the set of Γ-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, Γ, our goal is to decide the four questions from Section 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?

2.3 The Programs in Figure 1 have Different Properties

Figure 1 shows five example programs; now we will discuss them in detail.

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

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

A top migration exists. Consider $\lambda x.x(\mathsf{succ}(x(\mathsf{true})))$. 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 $\mathsf{Dyn} \to \mathsf{Dyn}$ and $\mathsf{Dyn} \to \mathsf{int}$. We have that $(\mathsf{Dyn} \to \mathsf{Dyn}) \sqsubseteq (\mathsf{Dyn} \to \mathsf{int})$. Thus, for $E = \lambda x.x(\mathsf{succ}(x(\mathsf{true})))$ and $\Gamma = \emptyset$, we have that $Mig_\Gamma(E)$ has a greatest element, namely the one that annotates x with $\mathsf{Dyn} \to \mathsf{int}$. In summary,

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

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

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

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 `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 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 THE SINGLETON PROBLEM

Our algorithm for the singleton problem relies on the static gradual guarantee (Theorem 2.3) 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 $(\texttt{Dyn} \to \texttt{Dyn})$.

$$\begin{aligned} \mathcal{S}(\texttt{bool}) &= \emptyset \\ \mathcal{S}(\texttt{int}) &= \emptyset \\ \mathcal{S}(\texttt{Dyn}) &= \{\ \texttt{bool},\ \texttt{int},\ \texttt{Dyn} \to \texttt{Dyn}\ \} \\ \mathcal{S}(T_1 \to T_2) &= \bigcup_{T_1' \in \mathcal{S}(T_1)} \{\ T_1' \to T_2\ \} \cup \bigcup_{T_2' \in \mathcal{S}(T_2)} \{\ T_1 \to T_2'\ \} \\ \\ \mathcal{S}(n) &= \emptyset \\ \mathcal{S}(\texttt{true}) &= \emptyset \\ \mathcal{S}(\texttt{false}) &= \emptyset \\ \mathcal{S}(x) &= \emptyset \\ \mathcal{S}(\lambda x : T.F) &= \bigcup_{T' \in \mathcal{S}(T)} \{\ \lambda x : T'.F\ \} \cup \bigcup_{F' \in \mathcal{S}(F)} \{\ \lambda x : T.F'\ \} \\ \mathcal{S}(E_1 E_2) &= \bigcup_{E_1' \in \mathcal{S}(E_1)} \{\ E_1' E_2\ \} \cup \bigcup_{E_2' \in \mathcal{S}(E_2)} \{\ E_1 E_2'\ \} \end{aligned}$$

For example,

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

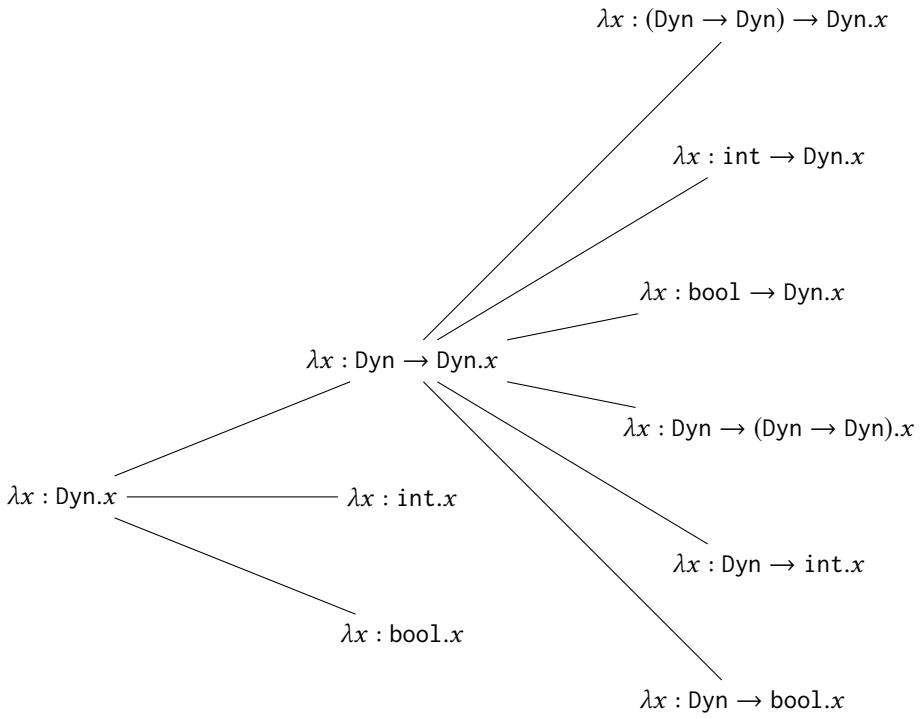

Fig. 3. The bottom three levels of the migration space for $\lambda x : \mathtt{Dyn}.x$.

Figure 3 shows the bottom three levels of the precision order for $\lambda x : \mathtt{Dyn}.x$. The idea is to call $\mathcal{S}(\lambda x : \mathtt{Dyn}.x)$ to obtain the second "column" of Figure 3. Additionally, we can call $\mathcal{S}(\lambda x : \mathtt{Dyn} \to \mathtt{Dyn}.x)$ to obtain the third "column" of Figure 3.

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

Theorem 3.1. $\forall T : \mathcal{S}(T) = \{\ T_u \mid T_u \neq T \ \wedge\ \forall T' : (T \sqsubseteq T' \sqsubseteq T_u)\ \textit{iff}\ ((T = T') \vee (T' = T_u))\ \}$.

Theorem 3.2. $\forall E : \mathcal{S}(E) = \{\ E_u \mid E_u \neq E \ \wedge\ \forall E' : (E \sqsubseteq E' \sqsubseteq E_u)\ \textit{iff}\ ((E = E') \vee (E' = E_u))\ \}$.

The proof of Theorem 3.1 is by straightforward induction on T, and the proof of Theorem 3.2 is by straightforward induction on E, using Theorem 3.1.

Theorem 3.3. $\forall E, \Gamma : \mathit{Mig}_\Gamma(E)$ *is a singleton iff* $\mathcal{S}(E) \cap \mathit{Mig}_\Gamma(E) = \emptyset$.

Proof. We will prove the two directions in turn.

Forwards direction. Suppose $\mathit{Mig}_\Gamma(E)$ is a singleton, that is $\mathit{Mig}_\Gamma(E) = \{E\}$. We have from Theorem 3.2 that $E \notin \mathcal{S}(E)$ so $\mathcal{S}(E) \cap \mathit{Mig}_\Gamma(E) = \emptyset$.

Backwards direction. Suppose (1) $\mathcal{S}(E) \cap \mathit{Mig}_\Gamma(E) = \emptyset$. Let (2) $E' \in \mathit{Mig}_\Gamma(E)$ 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 2.4 we have that (6) $\{\ E'' \mid E \sqsubseteq E'' \sqsubseteq E'\ \}$ is finite. From (6) and Theorem 3.2, 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 2.3), we have that (9) $\exists T'' : \Gamma \vdash E'' : T''$. From (8) and (9), we have that (10) $E'' \in \mathit{Mig}_\Gamma(E)$. 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, Γ, 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 3.4. *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 3.3 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 3.5. *We can solve the singleton problem in* $O(n^2)$ *time.*

Proof. We have from Theorem 3.4 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. □

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.

4.1 Constraints

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

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

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 φ to range over mappings from a finite set of type variables to types. We use $Dom(\varphi)$ to denote the domain of φ. For a type expression τ, we use $\varphi(\tau)$ to denote τ 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 φ 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.

4.2 Generating Constraints

From E, Γ, we generate constraints $Gen(E, \Gamma) \in PMEC$ as follows. Assume that E has been α-converted so that all bound variables are distinct from each other and distinct from the free variables. Let X be the set of λ-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 Γ: $Dom(\emptyset) = \emptyset$ 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 4.1. $\forall E, \Gamma$: *if* $FV(E) \subseteq Dom(\Gamma)$, *then* $(Mig_\Gamma(E), \sqsubseteq)$ *and* $(Sol(Gen(E, \Gamma)), \leq)$ *are order-isomorphic.*

We prove Theorem 4.1 in Appendix B of the supplementary material.

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:

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

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

$$SimPrec \quad : \quad PMEC \to MEC$$

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 4.2. $\forall A \in PMEC : Sol(A) = Sol(SimPrec(A))$.

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

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

Theorem 4.4. $\forall A \in MEC : Sol(A)$ *is finite iff* $\forall S \subseteq match(A) : Sol(SimMatch(A, S))$ *is finite.*

Proof. Immediate from Theorem 4.3. □

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 σ, σ' 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 4.5.

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

where $\sigma = Unify(A)$.

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

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

Theorem 4.6. *$\forall A \in EC : Sol(A)$ is finite iff ($\sigma \neq fail$ implies $Sol(SimEq(A, \sigma))$ is finite), where $\sigma = Unify(A)$.*

Proof. Immediate from Theorem 4.5. □

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

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

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
$\texttt{bool} \sim \texttt{bool}$	$\emptyset$
$\texttt{int} \sim \texttt{int}$	$\emptyset$
$\tau \sim \texttt{Dyn}$	$\emptyset$
$\texttt{Dyn} \sim \tau$	$\emptyset$
$(\tau_1 \rightarrow \tau_2) \sim \texttt{bool}$	(fail)
$(\tau_1 \rightarrow \tau_2) \sim \texttt{int}$	(fail)
$\texttt{bool} \sim (\tau_1 \rightarrow \tau_2)$	(fail)
$\texttt{int} \sim (\tau_1 \rightarrow \tau_2)$	(fail)
$\texttt{bool} \sim \texttt{int}$	(fail)
$\texttt{int} \sim \texttt{bool}$	(fail)
$(\tau_1 \rightarrow \tau_2) \sim (\tau_1' \rightarrow \tau_2')$	$\{\ (\tau_1 \sim \tau_1'),\ \ (\tau_2 \sim \tau_2')\ \}$
$\tau \sim \upsilon$	$\{\ \upsilon \sim \tau\ \}$

Theorem 4.7.

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

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 τ, we can create a syntax tree in which every node is labeled by a member of $\{\texttt{Dyn}, \texttt{bool}, \texttt{int}, \rightarrow\} \cup \mathit{TypeVar}$. For each node in the syntax tree for τ, we can consider the path α that leads from the root to that node. We use $\tau(\alpha)$ to denote label of the node reached by α. We define $\mathit{paths}(\tau)$ to be the set of paths from the root of τ to all leafs of τ. We have that τ 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.

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

Intuitively, $\mathit{BoundedVar}(\upsilon, T, A)$ says that "the variable υ 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 $(\upsilon \sim \tau')$ such that τ' has the same type as T at the corresponding leaf. For example, suppose $A = (V, S)$, where

$$\begin{aligned} V &= \{\ \upsilon, \upsilon_1, \upsilon_2\ \} \\ S &= \{\ \upsilon \sim (\texttt{bool} \rightarrow \upsilon_1),\ \ \upsilon \sim (\upsilon_2 \rightarrow \texttt{int})\ \} \end{aligned}$$

We have $\mathit{BoundedVar}(\upsilon, T, A)$, where $T = \texttt{bool} \rightarrow \texttt{int}$. 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 $\upsilon \sim (\texttt{bool} \rightarrow \upsilon_1)$, where $(\texttt{bool} \rightarrow \upsilon_1)$ has the same type (`bool`) as T at the corresponding leaf. Similarly, for the leaf `int` of T. we have in A the constraint $\upsilon \sim (\upsilon_2 \rightarrow \texttt{int})$, where $(\upsilon_2 \rightarrow \texttt{int})$ has the same type (`int`) as T at the corresponding leaf.

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

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

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

Theorem 4.8. *For $A \in C^{-}$: $Sol(A)$ is finite iff $Bounded(A)$.*

Proof. In the forwards direction, suppose $Sol(A)$ is finite. Thus, we can pick a maximal $\varphi \in Sol(A)$. Let $v \in vars(A)$. We will define a T such that $paths(T) = paths(\varphi(v))$. Let $\alpha \in paths(\varphi(v))$. Given that φ 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 α to be $\tau'(\alpha)$. As a result, we have $T \in MaximalType$ and $BoundedVar(v, T, A)$. We conclude $Bounded(A)$.

In the backwards direction, suppose $Bounded(A)$. For each variable, we have a lower bound and upper bound on the types that we can assign that variable. So, from Theorem 2.4 we have that $Sol(A)$ is finite. □

Theorem 4.9. *For $A \in C^{-}$, we can run $Bounded(A)$ in polynomial time.*

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

First we collect all constraints in A of the form $(v \sim \tau')$. Let $\bigcup_{\tau'} paths(\tau')$ denote the union of $paths(\tau')$ across all such τ'. Notice that we can construct this union in polynomial time. We see that $\bigcup_{\tau'} paths(\tau')$ defines the largest potential tree shape of T. For each $\alpha \in \bigcup_{\tau'} paths(\tau')$ 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 $\bigcup_{\tau'} paths(\tau')$ and determine whether any subset of $\bigcup_{\tau'} paths(\tau')$ defines a $T \in MaximumType$. This traversal can be done in polynomial time.

In summary, for each of the polynomially many $v \in vars(A)$, 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, Γ, where $FV(E) \subseteq Dom(\Gamma)$.
Problem: Is $Mig_{\Gamma}(E)$ finite?
Method:

```
1.  PMEC A1 = Gen(E, Γ)
2.  MEC A2 = SimPrec(A1)
3.  boolean finite = true
4.  for (S ⊆ match(A2)) {
5.      EC A5 = SimMatch(A2, S)
6.      (Subst ∪ {fail}) σ = Unify(A5)
7.      if (σ ≠ fail) {
8.          C A8 = SimEq(A5, σ)
9.          (C⁻ ∪ {fail}) A9 = SimCon(A8)
10.         if (A9 ≠ fail) {
11.             if ¬Bounded(A9) {
12.                 finite = false
13.             }
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 4.10. *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 4.1 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 4.2 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 4.4 $Sol(A_2)$ 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 4.6 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 4.7 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 4.8 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 4.11. *We can solve the finiteness problem in EXPTIME.*

Proof. We have from Theorem 4.10 that Algorithm **Finiteness Checker** is correct. Let us analyze the algorithm's time complexity. Let n be the total size of E and Γ. 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 4.9. 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. □

4.4 Example of how our Finiteness Checker works: $\lambda x.x(\mathsf{succ}(x(\mathsf{true})))$

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

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

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

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 succ, we can use that $\Gamma(\mathtt{succ}) = \mathtt{int} \to \mathtt{int}$.

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

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

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:

$$1 \;:\; [\![x]\!] \rhd \langle \mathtt{succ}(x(\mathtt{true}))\rangle \to [\![x(\mathtt{succ}(x(\mathtt{true})))]\!]$$
$$2 \;:\; [\![\mathtt{succ}]\!] \rhd \langle x(\mathtt{true})\rangle \to [\![\mathtt{succ}(x(\mathtt{true}))]\!]$$
$$3 \;:\; [\![x]\!] \rhd \langle \mathtt{true}\rangle \to [\![x(\mathtt{true})]\!]$$

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 $SimMatch(A_{10}, \{1, 2, 3\})$:

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

Notice that the only change from A_{10} to $SimMatch(A_{10}, \{1, 2, 3\})$ is that three occurrences of $\triangleright$ 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 (φ_{123}), where p, q are type variables.

$$\begin{array}{rcl} v & : & \varphi_{123}(v) \\ \hline [\![\lambda x.x(\mathtt{succ}(x(\mathtt{true})))]\!] & : & (p \to q) \to q \\ x & : & p \to q \\ [\![x(\mathtt{succ}(x(\mathtt{true})))]\!] & : & q \\ [\![x]\!] & : & p \to q \\ \langle \mathtt{succ}(x(\mathtt{true}))\rangle & : & p \\ [\![\mathtt{succ}]\!] & : & \mathtt{int} \to \mathtt{int} \\ \langle x(\mathtt{true})\rangle & : & \mathtt{int} \\ [\![\mathtt{succ}(x(\mathtt{true}))]\!] & : & \mathtt{int} \\ \langle \mathtt{true}\rangle & : & p \\ [\![x(\mathtt{true})]\!] & : & q \\ [\![\mathtt{true}]\!] & : & \mathtt{bool} \end{array}$$

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

$$\begin{array}{rcl} \langle \mathtt{succ}(x(\mathtt{true}))\rangle & \sim & [\![\mathtt{succ}(x(\mathtt{true}))]\!] \\ \langle x(\mathtt{true})\rangle & \sim & [\![x(\mathtt{true})]\!] \\ \langle \mathtt{true}\rangle & \sim & [\![\mathtt{true}]\!] \end{array}$$

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

$$\begin{array}{rcl} p & \sim & \mathtt{int} \\ \mathtt{int} & \sim & q \\ p & \sim & \mathtt{bool} \end{array}$$

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 $\mathtt{int} \sim q$ into $q \sim \mathtt{int}$:

$$\begin{array}{rcl} p & \sim & \mathtt{int} \\ q & \sim & \mathtt{int} \\ p & \sim & \mathtt{bool} \end{array}$$

Let us use A_{cm} to denote the above set of 3 Consistency constraints. We observe that $Bounded(A_{cm})$. Now we use Theorem 4.8 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 $SimMatch(A_{10}, S)$ contains an unsatisfiable subset of Equality constraints. So, for each of those cases, $Sol(SimMatch(A_{10}, S)) = \emptyset$.

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

$\lambda x.x(\texttt{succ}(x(\texttt{true})))$	$[\![\lambda x.x(\texttt{succ}(x(\texttt{true})))]\!] = x \to [\![x(\texttt{succ}(x(\texttt{true})))]\!]$
$x(\texttt{succ}(x(\texttt{true})))$	$[\![x]\!] = \texttt{Dyn}$
	$\langle \texttt{succ}(x(\texttt{true}))\rangle = \texttt{Dyn}$
	$[\![x(\texttt{succ}(x(\texttt{true})))]\!] = \texttt{Dyn}$
	$\langle \texttt{succ}(x(\texttt{true}))\rangle \sim [\![\texttt{succ}(x(\texttt{true}))]\!]$
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 $SimMatch(A_{10}, \{2\})$. Notice that $SimMatch(A_{10}, \{2\})$ has 11 Equality constraints. Those 11 Equality constraints are satisfiable and have the following most general unifier (φ_2), where p, q are type variables:

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

Next we apply φ_2 to A', The result is that $\mathit{SimEq}(\mathit{SimMatch}(A_{10}, \{2\}), \varphi_2)$ is:

$$\begin{array}{lcl} \mathsf{Dyn} & \sim & \mathsf{int} \\ \mathsf{int} & \sim & \mathsf{Dyn} \\ \mathsf{Dyn} & \sim & \mathsf{bool} \end{array}$$

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 $\mathit{SimCon}(A_2) = \emptyset$. Finally we observe that $\mathit{Bounded}(\emptyset)$. Now we use Theorem 4.8 to conclude that $\mathit{Sol}(\emptyset)$ is finite.

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

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

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

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

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

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 $[\![x]\!] = \mathtt{Dyn}$ and $[\![x]\!] = \langle \mathtt{true} \rangle \to [\![x(\mathtt{true})]\!]$. 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.

4.5 Example of how our Finiteness Checker works: $\lambda x.xx$

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

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

$\lambda x.xx$	$[\![\lambda x.xx]\!] = x \to [\![xx]\!]$
	$\mathtt{Dyn} \sqsubseteq x$
xx	$[\![x]\!] \rhd \langle x \rangle \to [\![xx]\!]$
	$\langle x \rangle \sim [\![x]\!]$
x	$[\![x]\!] = x$
x	$[\![x]\!] = x$

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 $\mathtt{Dyn} \sqsubseteq x$, which leaves us with the following 4 constraints.

$\lambda x.xx$	$[\![\lambda x.xx]\!] = x \to [\![xx]\!]$
xx	$[\![x]\!] \rhd \langle x \rangle \to [\![xx]\!]$
	$\langle x \rangle \sim [\![x]\!]$
x	$[\![x]\!] = x$

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 $SimMatch(A_4, S)$ has finitely many solutions.

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

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

Next we apply *SimEq* to $SimMatch(A_4, \{1\})$. Notice that $SimMatch(A_{10}, \{1\})$ has 3 Equality constraints. Those 3 Equality constraints are satisfiable and have the following most general unifier (φ_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 φ_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 4.8 to conclude that $Sol(A_1)$ is infinite.

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

5 THE TOP-CHOICE PROBLEM

The top-choice problem is: given E, Γ, 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 5.1. *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 2.4 that $\{ E' \mid E \sqsubseteq E' \sqsubseteq E_g \}$ is finite so also $Mig_\Gamma(E)$ is finite. □

Given Theorem 5.1, 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 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.

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

Algorithm: **Top-Choice Checker**.
Instance: E, Γ, where $FV(E) \subseteq Dom(\Gamma)$.
Problem: Does $Mig_\Gamma(E)$ have a greatest element?
Method:

```
1. int numMax = 0
2. if (Mig_Γ(E) is finite) {
3.         2^Terms workset = {E}
4.         2^Terms done = ∅
5.         while (workset ≠ ∅) {
6.                 pick E′ ∈ workset
7.                 remove E′ from workset and add E′ to done
8.                 if (∃T′ : Γ ⊢ E′ : T′) {
9.                         if (Mig_Γ(E′) is a singleton) {
10.                                numMax = numMax + 1
11.                        } else {
12.                                add (S(E′) \ done) to workset
13.                        }
14.                }
15.        }
16.}
17.return (numMax == 1)
```

Theorem 5.2. *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 5.3. *We can solve the top-choice problem in EXPTIME.*

Proof. We have from Theorem 5.2 that Algorithm **Top-Choice Checker** is correct. We have from Section 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. □

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.

6.1 A Semi-algorithm for the Maximality Problem

We can adapt the top-choice checker in Section 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.

6.2 The Maximality Problem is NP-hard

Theorem 6.1. *The maximality problem is NP-hard.*

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

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

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 λ-term E_F and type environment Γ_F:

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

The environment Γ_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 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 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 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 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 D 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 bool and that $\bar{x}_k$ is int.

Forwards direction. Suppose F is satisfiable and let ψ be a solution of F. Define φ 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 φ is a maximal solution of $Gen(E_F, \Gamma)$. First we show that φ 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 φ is maximal.

Consider x_k. Notice that while one of $\varphi(x_k)$ and $\varphi(\bar{x}_k)$ is Dyn, we cannot replace that 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 φ. Define a mapping ψ as follows.

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

We will show that ψ satisfies F. Consider the i'th clause of F. Given that φ 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 φ is a maximal solution of $Gen(E_F, \Gamma)$, we know that, for each x, the mapping φ 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 ψ 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 ψ satisfies the i'th clause. □

6.3 Example of how the NP-hardness Proof works

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

From F_2, we construct the following λ-term E_{F_2} and type environment Γ:

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

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 $\mathit{Gen}(E_{F_2}, \Gamma)$ has a maximal solution. Here are the three interesting subsets of $\mathit{Gen}(E_{F_2}, \Gamma)$:

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

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 ψ defined as follows: $\psi(x_1) = \texttt{true}$ and $\psi(x_2) = \texttt{true}$ and $\psi(x_3) = \texttt{false}$. From ψ we get φ:

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

We can check easily that $\varphi \models \mathit{Gen}(E_{F_2}, \Gamma)$. Additionally, we can check that φ is a maximal solution. Let us check every use of 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.

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

Fig. 4. Our benchmarks. Legend: ✓ means *yes*, and × means *no*, and ? means *unknown*.

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 6.3.

First, if we try other satisfying assignments of F than the ψ that we used in the example, we get other maximal solutions of $Gen(E, \Gamma)$ that are different from ψ. So, $Mig_\Gamma(E_F)$ 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 φ that assigns $\varphi(x_1) = \varphi(\bar{x}_1) = \varphi(x_2) = \varphi(\bar{x}_2) = \varphi(x_3) = \varphi(\bar{x}_3) = \mathtt{Dyn}$. This part of the definition of φ satisfies most of the constraints in $Gen(E, \Gamma)$ and leaves only $r_1 \sim (r_1 \to \mathtt{int})$ and $r_2 \sim (r_2 \to \mathtt{int})$. However, those constraints have infinitely many solutions. So, $Mig_\Gamma(E_F)$ has infinitely many solutions, hence the proof is of no help with proving a lower bound for finiteness problem.

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.

Benchmarks. Figure 4 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 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 [Giannini and Rocca 1988, 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 [Mogensen 1992, Section 3]. It uses the abbreviation $Y = \lambda h.(\lambda x.h(xx))(\lambda x.h(xx))$.

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

Benchmark	Singleton?	Top Choice?	Finite?	Has Max?
$\lambda x.x(\mathsf{succ}(x))$	326 ± 18ns	131 ± 2 µs	126 ± 5 µs	545 ± 12ns
$\lambda x.x(\mathsf{succ}(x(\mathsf{true})))$	147 ± 5 ns	313 ± 6 µs	296 ± 4 µs	3 ± 1 µs
$\lambda x. + (x\ 4)(x\ \mathsf{true})$	168 ± 3 ns	533 ± 14µs	517 ± 12µs	3 ± 1 µs
$(\lambda x.x)4$	132 ± 6 ns	35 ± 1 µs	33 ± 1 µs	531 ± 32ns
$\mathsf{succ}((\lambda y.y)((\lambda x.x)\mathsf{true}))$	335 ± 3 ns	209 ± 4 µs	196 ± 4 µs	2 µs
$\lambda x.x$	46 ± 3 ns	6 ± 1 µs	6 µs	371 ± 20ns
$\lambda x.\lambda y.yxx$	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 ± 15ns	587 ± 14µs	586 ± 17µs	5 s

Fig. 5. Execution times.

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

Fig. 6. Our tool's output of maximal migrations.

Execution. We ran each of our tools 100 times or more on each benchmark. Figure 5 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 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 6 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 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 paper by Campora et al. [2018]. 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`,

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

Fig. 7. The types for the entire program produced by the tool from [Campora et al. 2018] and by our tool.

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 7. 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 [Campora et al. 2018] produces a type anyway. The differences highlight that the tool from [Campora et al. 2018] may produce non-maximal migrations.

The reduction. We have implemented the reduction in Section 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 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.

8 RELATED WORK

We will discuss related work on type migration, type inference, and gradual typing design.

Type migration. The most closely related work is the POPL 2018 paper by Campora et al. [2018], which presented an efficient approach to migrating a program, but did not address our four decision problems. We saw in Section 7 that the approach in Campora et al. [2018] may produce non-maximal migrations; we will give an example of this below. The approach of Campora et al. [2018] integrates gradual types and variational types. Specifically, for each λ-bound variable, the approach uses constraints and unification to produce a static variational type that potentially can replace $\mathtt{Dyn}$. If the program has n such variables, those choices between $\mathtt{Dyn}$ and a static type create a finite migration space of size 2^n, which the approach searches efficiently.

For example, consider the program $(\lambda x : \mathtt{Dyn}.xx)$, which has no maximal migration. The constraint generation procedure generates a constraint of the form $\alpha \approx_\top d\langle \mathtt{Dyn}, \alpha \rangle \to \beta$. Here,

α, β are type variables, $d\langle \mathsf{Dyn}, \alpha \rangle$ is a type that signals a choice between Dyn and α, and $\approx_\top$ is a relationship that must be established via unification. The unification procedure finds that if we pick α, then $\alpha = \alpha \to \beta$ has no solution, so the approach picks Dyn. As a final step, the approach converts β to Dyn, and outputs the type (Dyn $\to$ Dyn) for α, which happens to be the type of the entire term (as shown in Figure 7). 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 $(\lambda x : \mathsf{Dyn}.xx)$ 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.

Siek and Vachharajani [2008] presented the first algorithm for type migration with gradual types. Their starting point was the type system by Siek and Taha [2006], for which they did type migration with a unification-based algorithm. Later, Garcia and Cimini [2015] presented a different unification-based algorithm for a similar type system. Both Siek and Vachharajani [2008] and Garcia and Cimini [2015] proved correctness, while neither had a report on experiments. Those two papers and our paper use similar forms of consistency constraints, but they differ in what questions they answer about such constraints. Specifically, Siek and Vachharajani [2008] and Garcia and Cimini [2015] focus on finding a single solution, while our paper studies properties of the set of solutions.

Rastogi et al. [2012] presented a migration algorithm for an object-oriented language with subtyping. They proved that the added types cannot cause new run-time failures.

Type inference. Type inference has a stricter goal than type migration, namely to infer static types for all variables, rather than to improve types as much as possible. We have two categories of type inference: static and dynamic. Static type inference includes the inference tool for Python by Hassan et al. [2018], and the inference tool for Dart by Heinze et al. [2016]. 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 by Xu et al. [2016]. Static type inference also includes the foundational work by Siek and Vachharajani [2008] and Garcia and Cimini [2015]. One property of the inference algorithm in [Garcia and Cimini 2015] 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. 2018]. The paper [Miyazaki et al. 2018] builds on the work of [Garcia and Cimini 2015] and proposed to delay the decision about what to do with the type variable until runtime, when those variables are instantiated.

Gradual typing design. Researchers have explored many interpretations and extensions of gradual types. Future work could pursue type migration that supports subtyping [Garcia et al. 2016; Siek and Taha 2007; Vitousek et al. 2014], refinement types [Lehmann and Éric Tanter 2017], monotonic references [Siek et al. 2015b], and polymorphism and set theoretic types [Castagna et al. 2019]. In particular, future work can consider our decision problems for those richer type systems.

9 CONCLUSION

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.

ACKNOWLEDGMENTS

We thank John Bender, Christian Kalhauge, Shuyang Liu, and Akshay Utture for helpful comments on a draft of the paper. We also thank Ben Greenman for providing one of the microbenchmarks.

REFERENCES

Gavin Bierman, Martín Abadi, and Mads Torgersen. 2014. Understanding TypeScript. In *ECOOP 2014 – Object-Oriented Programming*, Richard Jones (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 257–281.

John Peter Campora, Sheng Chen, Martin Erwig, and Eric Walkingshaw. 2018. Migrating Gradual Types. *Proc. ACM Program. Lang.* 2, POPL (2018), 15:1–15:29.

Giuseppe Castagna, Victor Lanvin, Tommaso Petrucciani, and Jeremy G. Siek. 2019. Gradual Typing: A New Perspective. *Proc. ACM Program. Lang.* 3, POPL, Article 16 (January 2019), 32 pages. https://doi.org/10.1145/3290329

Matteo Cimini and Jeremy Siek. 2016. The Gradualizer: A Methodology and Algorithm for Generating Gradual Type Systems. In *Proceedings of POPL'16, ACM Symposium on Principles of Programming Languages*.

Ronald Garcia and Matteo Cimini. 2015. Principal Type Schemes for Gradual Programs. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 303–315.

Ronald Garcia, Alison M. Clark, and Éric Tanter. 2016. Abstracting Gradual Typing. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 429–442.

Paola Giannini and Simona Ronchi Della Rocca. 1988. Characterization of Typings in Polymorphic Type Discipline. In *Proceedings of LICS'88, Third Annual Symposium on Logic in Computer Science*. 61–70.

Mostafa Hassan, Caterina Urban, Marco Eilers, and Peter Müller. 2018. MaxSMT-Based Type Inference for Python 3. In *Proceedings of CAV'18, Computer-Aided Verification*.

Thomas S. Heinze, Anders Møller, and Fabio Strocco. 2016. Type Safety Analysis for Dart. In *DLS*.

Nico Lehmann and Éric Tanter. 2017. Gradual Refinement Types. In *Proceedings of POPL, ACM Symposium on Principles of Programming Languages*.

Yusuke Miyazaki, Taro Sekiyama, and Atsushi Igarashi. 2018. Dynamic Type Inference for Gradual Hindley-Milner Typing. *CoRR* abs/1810.12619 (2018). arXiv:1810.12619 http://arxiv.org/abs/1810.12619

Torben Æ. Mogensen. 1992. Efficient Self-Interpretations in Lambda Calculus. *Journal of Functional Programming* 2, 3 (1992), 345–363. See also DIKU Report D–128, Sep 2, 1994.

Aseem Rastogi, Avik Chaudhuri, and Basil Hosmer. 2012. The Ins and Outs of Gradual Type Inference. *SIGPLAN Not.* 47, 1 (2012), 481–494.

Jeremy Siek and Walid Taha. 2007. Gradual Typing for Objects. In *Proceedings of the 21st European Conference on Object-Oriented Programming*. 2–27.

Jeremy G. Siek and Walid Taha. 2006. Gradual Typing for Functional Languages. In *IN SCHEME AND FUNCTIONAL PROGRAMMING WORKSHOP*. 81–92.

Jeremy G. Siek and Manish Vachharajani. 2008. Gradual Typing with Unification-based Inference. In *Proceedings of the 2008 Symposium on Dynamic Languages*. 7:1–7:12.

Jeremy G. Siek, Michael M. Vitousek, Matteo Cimini, and John Tang Boyland. 2015a. Refined Criteria for Gradual Typing. In *SNAPL*. 274–293.

Jeremy G. Siek, Michael M. Vitousek, Matteo Cimini, Sam Tobin-Hochstadt, and Ronald Garcia. 2015b. Monotonic References for Efficient Gradual Typing. In *Proceedings of the 24th European Symposium on Programming on Programming Languages and Systems - Volume 9032*. 432–456.

Sam Tobin-Hochstadt and Matthias Felleisen. 2008. The Design and Implementation of Typed Scheme. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '08)*. ACM, New York, NY, USA, 395–406. https://doi.org/10.1145/1328438.1328486

Michael M. Vitousek, Andrew M. Kent, Jeremy G. Siek, and Jim Baker. 2014. Design and Evaluation of Gradual Typing for Python. *SIGPLAN Not.* 50, 2 (2014), 45–56.

Zhaogui Xu, Xiangyu Zhang, Lin Chen, Kexin Pei, and Baowen Xu. 2016. Python probabilistic type inference with natural language support. In *Proceedings of 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*. 607–618.