Migrating Gradual Types

JOHN PETER CAMPORA, University of Louisiana at Lafayette, USA
SHENG CHEN, University of Louisiana at Lafayette, USA
MARTIN ERWIG, Oregon State University, USA
ERIC WALKINGSHAW, Oregon State University, USA

Gradual typing allows programs to enjoy the benefits of both static typing and dynamic typing. While it is often desirable to migrate a program from more dynamically-typed to more statically-typed or vice versa, gradual typing itself does not provide a way to facilitate this migration. This places the burden on programmers who have to manually add or remove type annotations. Besides the general challenge of adding type annotations to dynamically typed code, there are subtle interactions between these annotations in gradually typed code that exacerbate the situation. For example, to migrate a program to be as static as possible, in general, all possible combinations of adding or removing type annotations from parameters must be tried out and compared.

In this paper, we address this problem by developing *migrational typing*, which efficiently types all possible ways of adding or removing type annotations from a gradually typed program. The typing result supports automatically migrating a program to be as static as possible, or introducing the least number of dynamic types necessary to remove a type error. The approach can be extended to support user-defined criteria about which annotations to modify. We have implemented migrational typing and evaluated it on large programs. The results show that migrational typing scales linearly with the size of the program and takes only 2–4 times longer than plain gradual typing.

CCS Concepts: • **Theory of computation** → *Type theory*;

Additional Key Words and Phrases: gradual typing, variational types, program migration

ACM Reference Format:
John Peter Campora, Sheng Chen, Martin Erwig, and Eric Walkingshaw. 2018. Migrating Gradual Types. *Proc. ACM Program. Lang.* 2, POPL, Article 15 (January 2018), 29 pages. https://doi.org/10.1145/3158103

1 INTRODUCTION

Gradual typing promises to combine the benefits of static and dynamic typing in a single language. In the original formulation by Siek and Taha [2006], the goal is to bring the documentation and safety of static typing to a dynamically typed language. In their formalization, function parameters have dynamic types by default but can be explicitly annotated with static types. The resulting type system provides the same safety guarantees as static typing for expressions using type-annotated variables, yet allows the flexibility of dynamic typing for expressions with unannotated variables.

Dually, one can start with a statically typed language with type inference (such as F#, SML, OCaml, or Haskell) and allow the programmer to add annotations for dynamic types where needed [Garcia and Cimini 2015; Siek and Vachharajani 2008]. A function parameter can be annotated with `Dyn`

Authors' addresses: John Peter Campora, CACS, University of Louisiana at Lafayette, USA, campora@louisiana.edu; Sheng Chen, CACS, University of Louisiana at Lafayette, USA, chen@louisiana.edu; Martin Erwig, School of EECS, Oregon State University, USA, erwig@oregonstate.edu; Eric Walkingshaw, School of EECS, Oregon State University, USA, walkiner@oregonstate.edu.

2475-1421/2018/1-ART15
https://doi.org/10.1145/3158103

(the type of dynamic code) when dynamically typed behavior is needed or when the programmer is unsure whether all definitions are type-correct but wants to test the runtime behavior.

1.1 Challenges Applying Gradual Typing

By integrating static and dynamic typing, gradual typing not only enjoys the benefits of both typing disciplines, but also suffers from their respective shortcomings. For example, statically typed parts of the code have more restricted expressiveness and may contain static type errors that yield cryptic error messages [Tobin-Hochstadt et al. 2017], while dynamically typed parts of the code may contain dynamic type errors that are not captured until after the software is deployed. More interestingly, combining statically and dynamically typed code together can raise new challenges, for example, Takikawa et al. [2016] address the challenge of performance degradation in sound gradual typing at the boundaries between statically typed and dynamically typed code.

Therefore, to fully realize the benefits of gradual typing, we also need the ability to *navigate* along a program's dynamic-static typing spectrum, in order to make it more static or more dynamic when and where the respective strengths of each are desired. Answering the following three questions will help harness the full power of gradual typing.

Q1. Can we make a gradually typed program as static as possible, to maximize the advantages of static typing, while maintaining its well-typedness to keep it executable?

Q2. Can we migrate a program to a more static state while keeping some user-indicated parts dynamic? Such parts may be indicated, for example, to reduce the granularity of boundaries between static and dynamic code during execution, in order to maintain performance.

Q3. Can we introduce as few dynamics types as possible to migrate an ill-typed program to a type correct one while still enjoying the benefits of static typing for the well-typed parts?

The answers to these questions are not obvious. Furthermore, if the answers are *yes*, it is not clear whether we can implement the operations suggested by the questions efficiently.

We illustrate the challenges regarding Q1 by considering the following program written in the calculus by Garcia and Cimini [2015] extended with Haskell functions and notations, where parameters annotated with `Dyn` have dynamic types and those without annotations are inferred to have static types. In the rest of the paper, we say these parameters are *dynamic* and *static*, respectively. This program is adapted from van Keeken [2006] for formatting rows of a table according to a given width by trimming long rows and padding short rows with empty spaces.

```
rowAtI headOrFoot (fixed::Dyn) (widthFunc::Dyn) (table::Dyn) (border::Dyn) (i::Dyn) =
    let widest = maximum (map length table)
        row = table !! i
        width = if fixed then widthFunc fixed else widthFunc widest
    in if headOrFoot
       then replicate (width + 2) border
       else border ++ take width (row ++ replicate (width-length row) ' ') ++ border
```

The local variable `width` represents the width of the table and is computed by the argument `widthFunc`, either by applying it to `fixed` if `fixed` is true, or to `widest`, the size of largest row in the `table`. The argument `border` is added to the beginning and end of each row and also used to generate the header or footer row when the Boolean argument `headOrFoot` is true. If we bind the variable `tbl` to a list of strings, we can then call `rowAtI` in many ways, such as `rowAtI False True (const 3) tbl "_" 0`, `rowAtI False False id tbl "_" 1`, and `rowAtI True False id tbl '_' 0`.

After some testing, suppose we want to migrate `rowAtI` to a version that is as static as possible by removing `Dyn` annotations. Removing `Dyn` annotations turns out to be much trickier than we may expect. First, if we remove all `Dyn` annotations, then type inference fails for `rowAtI` since it contains

multiple static type errors, for example, the then branch requires `border` to have type `Char` while the else branch requires it to have type `[Char]`. Second, if we remove `Dyn` annotations in a left-to-right order, we will encounter a type error as soon as the annotation for `widthFunc` is removed. (In this paper, we follow the spirit of Garcia and Cimini [2015] to infer static types only.) However, this does not necessarily indicate that the error was solely caused by `widthFunc` being statically typed. In fact, the type error involving `widthFunc` is due to the interaction with `fixed` when computing the value of `width`. At this point, we can restore the well-typedness of `rowAtI` by *either* re-annotating `fixed` *or* `widthFunc` with `Dyn`. Unfortunately, we cannot easily gauge which annotation is better for typing the rest of the function. If we choose to re-annotate `fixed`, we will encounter another type error when the `Dyn` annotation for `border` is removed. Does this type error go away if we instead mark `fixed` as static and `widthFunc` as dynamic? The easiest way to tell is by trying it out.

The example illustrates that parameters give rise to complicated typing interactions. The type error caused by making one parameter static may be avoided by making another parameter dynamic, or the type error caused by making two parameters static can be fixed by making another dynamic, and so on. In general, we must examine all possible combinations of static vs. dynamic parameters to identify a program that is both well typed and as static as possible. We refer to all of the potential programs produced by adding or removing `Dyn` annotations as a *migration space*. We say a program in the migration space has a *most static type* if removing any `Dyn` from the program will make it ill typed. We call a migration that yields a program with a most static type a *most static migration*. Due to the nature of type interactions, the most static type, and thus the most static migration, is not unique. Since every parameter can be either static or dynamic, the size of the migration space is exponential in the number of parameters for all functions in the program. For the program consisting of only `rowAtI`, which has six parameters, we would need to try out all $2^6 = 64$ combinations to identify the most static migrations.

Questions Q2 and Q3 are similarly difficult for the same underlying reason that the typing of different parameters are interrelated. This quality of type inference precludes the possibility of a greedy algorithm that considers each parameter in turn, adding or removing `Dyn` annotations. In general, we must conceptually explore all of the possibilities in the migration space.

The challenges posed by migration between more and less static programs may prevent programmers from fully realizing the potential of gradual type systems. As evidence for this, the CircleCI project recently abandoned Typed Clojure mainly because the cost of adding type annotations to Clojure programs was perceived to exceed the benefits.[1] Similarly, Tobin-Hochstadt et al. [2017] reported that migration of Racket modules to Typed Racked requires too much effort.

1.2 Migrating Gradual Types

In this paper, we address Q1 by: (1) developing a type system that efficiently types the entire migration space and (2) designing a method to traverse the result of typing the migration space, calculating which `Dyn` annotations can be removed. In this paper, we mainly consider the *removal* of `Dyn` annotations to support migrating to a more statically typed program; that is, we make types more precise [Siek and Taha 2006]. However, in Section 8, we describe how the approach can be extended to support the addition of `Dyn` annotations, along with extensions to address Q2 and Q3.

As demonstrated in Section 1.1, in general, finding the most static migration requires exploring the entire migration space, which is exponential in size. This rules out a simple brute-force approach that type checks each possibility and compares the results to find the best one.

To illustrate how we can improve on a brute-force search, let us focus on a single parameter, say `i` in the `rowAtI` function from Section 1.1. To decide whether we can remove the `Dyn` annotation, we

[1] https://circleci.com/blog/why-were-no-longer-using-core-typed/

Program	Dyn annotations	Type for rowAtI
1	+ + + + +	Bool → Dyn → Dyn → Dyn → Dyn → Dyn → [Char]
2	− + + + +	Bool → Bool → Dyn → Dyn → Dyn → Dyn → [Char]
3	− + − + −	Bool → Bool → Dyn → [[Char]] → Dyn → Int → [Char]
4	+ − + + +	Bool → Dyn → (Int→Int) → Dyn → Dyn → Dyn → [Char]
5	+ − − + −	Bool → Dyn → (Int→Int) → [[Char]] → Dyn → Int → [Char]
6	− − + + +	✗
7	+ + + − +	✗
8	+ + − − −	✗

Fig. 1. Types for a sample of the migration space for the rowAtI function. The second column contains a sequence of + and − symbols, indicating whether the Dyn annotation is kept or removed, respectively, for each of the five parameters annotated with Dyn in rowAtI. For example, for program 2, all parameters except fixed keep their Dyn annotations. The ✗ entries denote that the corresponding program is ill typed.

need to type two programs: one where i is static and one where i is dynamic. Observe that the two typing processes differ only slightly. Of the three let-bound variables, only the type of the second (row) is affected by whether i is static or dynamic. The typing of the other two let-bound variables is identical in both cases. Moreover, since the type of row is determined to be the same regardless of whether i is static or dynamic, the typing of the body of the let-expression is also identical.

This observation suggests that we should reuse typing results while exploring the migration space to determine which Dyn annotations can be removed. A systematic way to support this reuse is provided by *variational typing* [Chen et al. 2012, 2014]. In this paper, we develop a type system that integrates gradual types [Siek and Taha 2006] and variational types [Chen et al. 2014] to support reuse when typing the migration space. This type system supports efficiently typing the entire migration space, in roughly linear time, despite the presence of type errors.

After typing the migration space, we want to find the point in that space that is most static. Although the number of results to be considered is large, this step can be made efficient by exploiting several of relationships between the resulting types. To illustrate these relationships, we list a subset of the migration space for the rowAtI example and their corresponding types in Figure 1.

The first observation is that some parameters, whether they are static or dynamic, do not affect the type correctness of the program. In the example, the 3rd and 5th parameters (table and i, respectively) are examples of such parameters. Given this knowledge and the fact that program 3 is well typed, we can deduce that program 2 is also well typed since they differ only in the Dyn annotations of the 3rd and 5th parameters. Similarly, given that program 8 is type incorrect, we can deduce that program 7 is also type incorrect for the same reason.

The second observation is that if a program is well typed after removing Dyn annotations from a set of parameters P, then (1) removing Dyn annotations from a subset of P will also yield a well-typed program (this corresponds to the static gradual guarantees of Siek et al. [2015]), and (2) the program with all Dyn annotations removed from P is the most statically typed of these programs. For example, program 3 has a more static type than program 2, which in turn has a more static type than program 1. Similarly, this relation holds for the sequence of programs 5, 4, and 1. Note that the number of removed Dyn annotations does not provide the same ordering. For example, program 3 removes more Dyn annotations than program 4, but program 4 has a more static type.

The third observation is that, if removing all Dyn annotations for a set of parameters causes a type error, then removing the Dyn annotations for any superset of those parameters must also cause a type error. For example, given that making the 4th parameter (border) static in program 7 causes

a type error, we can deduce that additionally making the 3rd (`table`) and 5th (`i`) parameters static in program 8 will also cause a type error.

These three observations enable an efficient method for finding the most static program. For `rowAtI`, we immediately discover that programs 3 and 5 are most static (neither one is more static than the other). In this case, we can either pick one of the results or have a programmer specify the preferable program. In Section 5, we show that these three observations hold for arbitrary programs, which allows us to develop an efficient method for finding desired programs in general.

We make the following contributions in this paper:

(1) In Section 1.1, we identify three questions, Q1 through Q3, for migrating gradual program to fully harness the benefits of gradual typing.
(2) In Section 4, we present a type system that integrates gradual types [Siek and Taha 2006], variational types [Chen et al. 2014], and error tolerant typing [Chen et al. 2012]. The type system is correct and efficiently types the whole migration space.
(3) In Section 5, we investigate the relationship between different candidate migrations and develop a method for computing the most static migrations.
(4) In Sections 6 and 7, we generate and solve constraints to provide type inference for migrational typing, and prove that the constraint solving algorithm is correct.
(5) In Section 8 we describe extensions to migrational typing to answer all of the questions outlined in Section 1.1, and to support additional common language features.
(6) In Section 9, we study the performance of our implementation by applying it to synthesized programs. The result shows that our approach scales linearly with program size.

To improve readability, the following table summarizes where important terms and operations are introduced. In the "F | P" column, F i and P i are shorthands for Figure i and Page i, respectively.

Term	Notation	F | P	Operation	Notation	F | P
static types	T	F 3	selection	$\lfloor\cdot\rfloor_{d.1}$	P 7
gradual types	G	F 3	compatibility (M)	$\approx$	F 4
variational types	V	F 3	constrained compatibility (M)	$\approx_\pi$	F 5
migrational types	M	F 3	constrained operation (M)	op_π	F 5
statifier	ω	F 2	better ordering (G)	$\preceq$	P 15
variational statifier	Ω	F 3	more static ordering (G)	$\sqsubseteq$	P 15
choices	$d\langle,\rangle$	P 7	stricter ordering (δ)	$\gg$	P 16
decisions/eliminators	δ	P 7/P 16	less defined ordering (π)	$\leq$	F 6
valid eliminators	δ^v	P 16	pattern meet (π)	$\otimes$	P 19
typing pattern	$\pi, \top, \bot$	F 5			

2 BACKGROUND AND PREPARATION

In this section, we briefly introduce two areas of previous work that our type system for migrating gradual types builds on. In Section 2.1, we present a simple gradually typed language that represents the starting point for our work. This language is adapted from Garcia and Cimini [2015], but includes some minor differences to set up the presentation in Section 4. In Section 2.2, we introduce the concept of variational typing [Chen et al. 2014], which is the key technique that allows us to efficiently type the entire migration space.

2.1 Gradual Typing

Gradual typing allows the interoperability of statically typed and dynamically typed code. The original formalization by Siek and Taha [2006] defined gradual typing for a simply typed lambda

Syntax:

$$\begin{array}{lrcl} \text{Expressions} & e & ::= & c \mid x \mid \lambda x.e \mid \lambda x : \texttt{Dyn}.e \mid e\ e \mid \textbf{if}\ e\ \textbf{then}\ e\ \textbf{else}\ e \\ \text{Static types} & T & ::= & \gamma \mid \alpha \mid T \to T \\ \text{Gradual types} & G & ::= & \gamma \mid \alpha \mid G \to G \mid \texttt{Dyn} \end{array}$$

Type system: $\boxed{\omega; \Gamma \vdash_{GC} e : G}$

$$\textsc{Con}\ \frac{c \text{ is of type } \gamma}{\omega; \Gamma \vdash_{GC} c : \gamma} \qquad \textsc{Var}\ \frac{x : G \in \Gamma}{\omega; \Gamma \vdash_{GC} x : G} \qquad \textsc{Abs}\ \frac{\omega; \Gamma, x \mapsto T \vdash_{GC} e : G}{\omega; \Gamma \vdash_{GC} \lambda x.e : T \to G}$$

$$\textsc{AbsDyn}\ \frac{\omega; \Gamma, x \mapsto \omega(x) \vdash_{GC} e : G'}{\omega; \Gamma \vdash_{GC} \lambda x : \texttt{Dyn}.e : \omega(x) \to G'} \qquad \textsc{App}\ \frac{\omega; \Gamma \vdash_{GC} e_1 : G \qquad \omega; \Gamma \vdash_{GC} e_2 : G' \qquad dom\,(G) \sim G'}{\omega; \Gamma \vdash_{GC} e_1\ e_2 : cod\,(G)}$$

$$\textsc{If}\ \frac{(\omega; \Gamma \vdash_{GC} e_i : G_i)^{i:1..3} \qquad \texttt{Bool} \sim G_1}{\omega; \Gamma \vdash_{GC} \textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3 : G_2 \sqcap G_3}$$

Gradual type consistency:

$$\text{C1}\ \ G \sim G \qquad \text{C2}\ \ G \sim \texttt{Dyn} \qquad \text{C3}\ \ \texttt{Dyn} \sim G \qquad \text{C4}\ \frac{G_{11} \sim G_{21} \qquad G_{12} \sim G_{22}}{G_{11} \to G_{12} \sim G_{21} \to G_{22}}$$

Auxiliary definitions:

$$\begin{array}{rl} dom\,(G_1 \to G_2) = G_1 & \qquad \texttt{Dyn} \sqcap G = G \\ dom\,(\texttt{Dyn}) = \texttt{Dyn} & \qquad G \sqcap \texttt{Dyn} = G \\ cod\,(G_1 \to G_2) = G_2 & \qquad G \sqcap G = G \\ cod\,(\texttt{Dyn}) = \texttt{Dyn} & \qquad G_{11} \to G_{12} \sqcap G_{21} \to G_{22} = (G_{11} \sqcap G_{21}) \to (G_{12} \sqcap G_{22}) \end{array}$$

Fig. 2. Syntax and type system of ITGL, an implicitly typed gradual language. The operations *dom*, *cod*, and $\sqcap$ are undefined for cases that are not listed here.

calculus extended with dynamic types. Siek and Vachharajani [2008] and Garcia and Cimini [2015] further investigated gradual typing in the presence of type inference.

In this paper, we consider the migration of programs in implicitly typed gradual languages. In Figure 2, we present the syntax and type system of one such language, ITGL, which is adapted from Garcia and Cimini [2015] and forms the basis for this work. In the syntax, c ranges over constant values, x over variables, γ over constant types, and α over type variables. There are two cases for abstraction expressions, one where the parameter is annotated by Dyn and one where it is not. The rest of the cases are standard. The type system will be explained below.

The presentation of ITGL in Figure 2 differs from the original in Garcia and Cimini [2015] in two ways. First, our syntax is more restrictive: we omit a case for explicit type ascription of expressions and we do not allow arbitrary type annotations on abstraction parameters. We also don't consider let-polymorphism here. These restrictions are made to simplify our formalization later, but we show in Section 8 how they can be lifted. Second, the typing rules are parameterized by a *statifier*, ω, which is used in the full migrational type system later (Section 4). The statifier specifies what static types to assign to parameters whose Dyn annotations will be removed. For simplicity, we assume parameters have unique names. In the type system as defined in Figure 2, ω is always empty and $\omega(x) = \texttt{Dyn}$ for any parameter x, corresponding to the type system in Garcia and Cimini [2015].

In the type system for ITGL in Figure 2, the typing rules for constants and variables are standard. There are two rules for abstractions, Abs for unannotated parameters which must have static types, and AbsDyn for annotated parameters which may have dynamic types. Typing applications is tricky since dynamically typed arguments can be passed to functions with statically typed parameters and vice versa. For example, assuming the function, `succ`, has static type `Int` → `Int`, both of the following programs in our Haskell-like notation should be accepted by gradual typing.

```
inc (num::Dyn) = succ num
foo (f::Dyn) = f True
```

The App rule accommodates this with the help of a *consistency* relation, ∼, that dictates when two unequal types are compatible with each other. An application is well typed if the domain of the LHS (i.e. the parameter type) is consistent with the RHS, and the type of the application is the codomain of LHS. The auxiliary functions *dom* and *cod* return the domain and codomain of a function type, respectively, or `Dyn` for a dynamic type (reflecting the fact that `Dyn` is equivalent to `Dyn` → `Dyn`).

The gradual type consistency relation is defined in Figure 2 by four rules: C1 defines that consistency is reflexive, C2 and C3 define that a dynamic type is consistent with any type, and C4 defines that two functions types are consistent if their respective argument and return types are consistent. As a result, `Int` → `Int` ∼ `Int` → `Dyn` but not `Int` → `Int` ∼ `Bool` → `Dyn`, since the argument types are not consistent in the latter case. Note that the consistency relation is not transitive. Due to C2 and C3, transitivity would lead every static type to be consistent with every other static type, which is clearly undesirable.

Typing conditional expressions relies on the meet operation, ⊓, on gradual types. Intuitively, meet chooses the more static of two base types when one is `Dyn`. For two equal static types, meet is idempotent. For two function types, meet is applied recursively to their respective argument and return types. The meet operation helps assign types to conditionals when the two branches might not have an identical type but still have consistent types. Intuitively, meet favors the type of the more static branch of the conditional expression.

2.2 Variational Typing

Variational typing [Chen et al. 2012, 2014] enables efficiently inferring types for *variational programs*. A variational program represents many different variant programs that share some parts amongst each other and which can each be generated through a static process of *selection*.

The theoretical foundation for variational typing is the choice calculus [Erwig and Walkingshaw 2011], a formal language for representing variational programs. The essence of the choice calculus is that static variability in programs can be locally captured in variation points called *choices*, as demonstrated by the following example.

$$\texttt{vfun} \;=\; A\langle \texttt{succ}, \texttt{even}\rangle \;\; \texttt{1}$$

This program contains a choice named A with two alternatives, `succ` and `even`. We write $\lfloor e \rfloor_{d.i}$ to indicate the selection of the ith alternative of each choice named d in e. So, $\lfloor \texttt{vfun} \rfloor_{A.1}$ yields the program `succ 1` and $\lfloor \texttt{vfun} \rfloor_{A.2}$ yields `even 1`. We call $d.i$ a selector and use s to range over selectors.

A *decision* is a set of selectors; we use δ to range over decisions. The elimination of choices extends naturally to decisions by selecting with each selector in the decision. An expression e is called *plain* if it does not contain any choices and is called *variational* if it does contain choices. A plain expression obtained by eliminating all choices in a variational expression is called a *variant*. For example, `succ 1` is a plain expression and a variant of the variational expression `vfun`.

A variational expression may contain several choices. Choices with the same name are synchronized and independent otherwise. For example, the variational expression $A\langle \texttt{succ}, \texttt{even}\rangle \;\; A\langle 2, 3\rangle$ has two variants, `succ 2` and `even 3`, obtained by the decisions $\{A.1\}$ and $\{A.2\}$, respectively. The

program `succ 3` *cannot* be obtained through selection and so is *not* a variant of this expression. On the other hand, the variational expression $A\langle \texttt{succ}, \texttt{even} \rangle\ B\langle 2, 3 \rangle$ has four variants, and we can obtain the variant `succ 3` with the decision $\{A.1, B.2\}$.

In general, an expression with n distinct choice names can be configured in 2^n different ways. Since variational programs can easily contain hundreds or thousands of independent choice names [Apel et al. 2016], checking the type correctness of all variants is intractable by a brute-force strategy of generating all of the variants and typing each one individually [Thüm et al. 2014]. Variational typing solves this problem by sharing the typing process across all variants, which is achieved by defining and reasoning about variational types.

Variational types are types extended with choices. All concepts and operations on variational expressions carry over to variational types. It is natural to assign variational types to variational expressions. For example, $A\langle \texttt{succ}, \texttt{even} \rangle$ has type $A\langle \texttt{Int} \to \texttt{Int}, \texttt{Int} \to \texttt{Bool} \rangle$. Similar to gradual typing, typing applications in the presence of variation is complicated by the fact that "compatible" types may not be syntactically equal. In particular, (1) the LHS is traditionally expected to be a function type but in variational typing may be a (nested) choice of function types, and (2) when checking whether the type of the argument matches the type of the parameter, we must take into account that either or both may be variational. For example, the type of the function on the LHS of `vfun` is $A\langle \texttt{Int} \to \texttt{Int}, \texttt{Int} \to \texttt{Bool} \rangle$, which is not a function type directly, but both variants of `vfun succ 1` and `even 1`, are well typed.

Typing applications is supported in variational typing through the definition of a type equivalence relation [Chen et al. 2014], which specifies when a type can be transformed into another without affecting its semantics. The semantics of a variational type maps decisions to the variant plain types obtained by selecting from the type using the decision. For example, $A\langle \texttt{Int} \to \texttt{Int}, \texttt{Int} \to \texttt{Bool} \rangle$, $A\langle \texttt{Int}, \texttt{Int} \rangle \to A\langle \texttt{Int}, \texttt{Bool} \rangle$, and $\texttt{Int} \to A\langle \texttt{Int}, \texttt{Bool} \rangle$ are all equivalent because selecting from each of them with $\{A.1\}$ yields the same type $\texttt{Int} \to \texttt{Int}$ and selecting from each of them with $\{A.2\}$ yields the same type $\texttt{Int} \to \texttt{Bool}$. As a result, we can say that `vfun` has the type $\texttt{Int} \to A\langle \texttt{Int}, \texttt{Bool} \rangle$, which is a function type with the argument type `Int` matching the type of `1`. We can thus assign the type $V_{\texttt{vfun}} = A\langle \texttt{Int}, \texttt{Bool} \rangle$ to `vfun`.

An important result of variational typing is that choice elimination preserves typing. More specifically, if e has the type V, then $\lfloor e \rfloor_\delta$ has the type $\lfloor V \rfloor_\delta$ for any decision δ. For example, $\lfloor \texttt{vfun} \rfloor_{A.1}$ yields `succ 1`, which has the type `Int`, the same as $\lfloor V_{\texttt{vfun}} \rfloor_{A.1}$. An implication of this result is that the type of any variant can be easily obtained by making an appropriate selection into the result type of the variational program. Another important result of variational typing is that it is significantly faster than the brute-force approach.

3 ROAD MAP TO MIGRATING GRADUAL TYPES

In Section 1.1, we argued that the complexity of the tasks implied by the questions Q1–Q3, involving the migration of gradual programs, is exponential. In Section 2.2, we have shown that variational typing can efficiently type a set of similar programs. A main idea of this paper is to reduce the problem of typing the migration space to variational typing. Specifically, we assign each parameter with a `Dyn` annotation a choice type whose the first alternative is a `Dyn` and whose second alternative is a static type. Consider, for example, the following function `widthV` that represents the variationally typed version of the function `width` (also shown below) for computing the table width in `rowAtI`.

```
width (fixed::Dyn) (widthFunc::Dyn) = if fixed then widthFunc fixed else widthFunc 5
widthV (fixed::A⟨Dyn,Bool⟩) (widthFunc::B⟨Dyn,Int → Int⟩) =
    if fixed then widthFunc fixed else widthFunc 5
```

The function `widthV` encodes all four possible migrations of `width`. If $V_{\texttt{widthV}}$ is the type of `widthV`, then $\lfloor V_{\texttt{widthV}} \rfloor_{\{A.1, B.1\}}$ is the type for `width` with no `Dyn` annotations removed, $\lfloor V_{\texttt{widthV}} \rfloor_{\{A.2, B.1\}}$ is the type that replaces `Dyn` with `Bool` for `fixed` and keeps `Dyn` for `widthFunc`, $\lfloor V_{\texttt{widthV}} \rfloor_{\{A.1, B.2\}}$ is the type that keeps `Dyn` for `fixed` but replaces `Dyn` with `Int` $\rightarrow$ `Int` for `widthFunc`, and $\lfloor V_{\texttt{widthV}} \rfloor_{\{A.2, B.2\}}$ is the type that removes both `Dyn` annotations.

In order to successfully employ variational typing to improve the performance of migrational typing, several technical challenges must be addressed.

C1. In the presence of dynamic and variational types, we need to combine the type equivalence relation and the consistency relation, which we refer to as the *compatibility* relation. After introducing the syntax of the migrational type system in Section 4.1, we address this problem in Section 4.2.

C2. In general, some variants of the variational program that encodes the migration space may contain type errors. We need the typing process to continue even in the presence of type errors to determine the types of all variants. In Section 4.3, we address this problem and give a declarative specification of our type system.

C3. In `widthV`, we explicitly assigned static types to each parameter. One may wonder whether these are the best types to assign. Maybe other static types could improve the typing result and produce more general types or fewer type errors. After presenting the typing rules in Section 4.4, we prove in Section 4.5 that in our type system, there exists a best typing derivation that contains the fewest errors and yields most static and general result types.

C4. With the best migrational typing, we have to determine the combination of `Dyn` removals that makes the program as static as possible. This may require the comparison of an exponential number of result types for the migration space. In Section 5, we develop an efficient algorithm for solving this problem.

C5. In challenge C3 we claimed that a best migrational typing exists, but how do we find it? We answer this question by solving the type inference problem in Sections 6 and 7.

4 MIGRATIONAL TYPE SYSTEM

This section addresses the challenges C1–C3 from Section 3 to support efficient migrational typing. After introducing the syntax of types and expressions in Section 4.1, the compatibility relation is defined in Section 4.2, addressing C1. A *pattern-constrained* typing relation is introduced in Section 4.3 and defined via typing rules in Section 4.4, addressing C2. Finally, the properties of this type system are discussed in Section 4.5, addressing C3.

4.1 Syntax

The syntax of expressions, types, and environments is given in Figure 3. The metavariables we use to range over the relevant symbol domains are listed at the top figure. For type variables, we typically use β to denote the result type of a function application during constraint generation and κ to denote fresh type variables generated during constraint generation and solving (see Sections 6 and 7). For choice names, we typically use A and B to denote arbitrary specific choices in examples and d as a generic metavariable to range over choices names in definitions.

The syntax of expressions, static types, and gradual types are repeated from Section 2.1. To this, we add variational types, which are static types extended with choices, and migrational types, which are gradual types extended with choices. Note that each top-level parameter is assigned a restricted form of migrational type, which is either a fully static type, a `Dyn`, or a choice of restricted migrational types; however, the more general syntax defined in Figure 3 is needed during the typing

Term variables	x, y, z	Value constants	c	Choice names	A, B, d
Type variables	α, β, κ	Type constants	γ	Program locations	l

Expressions	e	::=	$c \mid x \mid \lambda x.e \mid \lambda x : \texttt{Dyn}.e \mid e\ e \mid \textbf{if}\ e\ \textbf{then}\ e\ \textbf{else}\ e$
Static types	T	::=	$\gamma \mid \alpha \mid T \to T$
Gradual types	G	::=	$\gamma \mid \alpha \mid G \to G \mid \texttt{Dyn}$
Variational types	V	::=	$\gamma \mid \alpha \mid V \to V \mid d\langle V, V\rangle$
Migrational types	M	::=	$\gamma \mid \alpha \mid M \to M \mid \texttt{Dyn} \mid d\langle M, M\rangle$
Type environment	Γ	::=	$\varnothing \mid \Gamma, x \mapsto M$
Substitution	θ	::=	$\varnothing \mid \theta, \alpha \mapsto V$
Variational statifier	Ω	::=	$\varnothing \mid \Omega, x \mapsto V$

Fig. 3. Syntax of expressions, types, and environments.

process. In Section 8.2, we extend our framework to allow an arbitrary mix of Dyn and static types for top-level parameters.

The type system relies on three kinds of environments: a type environment maps variables to migrational types, a substitution maps type variables to variational types, and a *variational statifier* maps variables to variational types. As described in Section 2.1, a statifier ω records one way of making a program more static (by removing some subset of Dyn annotations). A variational statifier Ω instead compactly encodes all possible statifiers for an expression. Since we want migration in our formalization to assign static types to parameters whose Dyn annotations are removed, Ω maps parameters to variational types, but not migrational types.

Substitutions map type variables to variational types rather than migrational types since substituting dynamic types is unsound. For example, suppose we have $\texttt{f} \mapsto \alpha \to \alpha \to \alpha \to \alpha$ and $x \mapsto \texttt{Dyn}$ in Γ. Now, when typing the application `f x`, we will substitute $\{\alpha \mapsto \texttt{Dyn}\}$, yielding $\texttt{Dyn} \to \texttt{Dyn} \to \texttt{Dyn}$ as the type of `f x`. However, this implies that `f x 2 True` is well typed, even though this violates the initial static type of `f`. Type substitution, written as $\theta(M)$, is defined in the conventional way.

4.2 Type Compatibility

In the rest of this section, we use the `widthV` example (Section 3) to motivate the technical development of the migration type system and investigate the properties of the type system. The motivating goal is to type the condition `fixed` and the application `widthFunc 5` in `widthV`.

According to the annotation of `widthV`, the parameter `fixed` has type $A\langle\texttt{Dyn}, \texttt{Bool}\rangle$. Since `fixed` is used as a condition, it should have type `Bool`. Since both alternatives of the choice are consistent with `Bool`, this use should be considered well typed. The variable `widthFunc` has type $B\langle\texttt{Dyn}, \texttt{Int} \to \texttt{Int}\rangle$, which can be considered equivalent to $B\langle\texttt{Dyn}, \texttt{Int}\rangle \to B\langle\texttt{Dyn}, \texttt{Int}\rangle$ (in Section 4.4, we show how to achieve this formally with *dom* and *cod*). The constant `5` has type `Int`. Since both alternatives of $B\langle\texttt{Dyn}, \texttt{Int}\rangle$ are consistent with `Int`, `widthFunc 5` should also be considered well typed.

These two examples demonstrate that we need a notion of *compatibility* between two migrational types to express that all of their variants are consistent. Intuitively, the compatibility relation incorporates both type equivalence for variational types [Chen et al. 2014] and type consistency for gradual types [Siek and Taha 2006]. The definition of compatibility ($M_1 \approx M_2$) is given in Figure 4. The relation is reflexive (T1) and symmetric (T2). The relation is transitive (T3) in the case that no Dyns are present, which we indicate by using the metavariable for variational types (V).

The rules T4 and T5 specify compatibility under choice type simplification. Rule T4 states that a choice with identical alternatives is compatible with its alternatives. Rule T5 says that two

$$\text{T1}\quad M \approx M \qquad \text{T2}\ \frac{M_1 \approx M_2}{M_2 \approx M_1} \qquad \text{T3}\ \frac{V_1 \approx V_2 \qquad V_2 \approx V_3}{V_1 \approx V_3}$$

$$\text{T4}\quad d\langle M, M\rangle \approx M \qquad \text{T5}\quad d\langle M_1, M_2\rangle \approx d\langle \lfloor M_1 \rfloor_{d.1}, \lfloor M_2 \rfloor_{d.2}\rangle$$

$$\textsc{Cong}\ \frac{M_1 \approx M_2}{M[M_1] \approx M[M_2]} \qquad \textsc{DynIntro}\ \frac{M_1 \approx M_2}{M_1 \approx M_2[\texttt{Dyn}]}$$

Fig. 4. Rules defining type compatibility

types are compatible under elimination of dead alternatives. Note that the operation $\lfloor M_1 \rfloor_{d.1}$ in the first alternative of d replaces each occurrence of a d choice in M_1 with its first alternative and thus removes the second alternative, which is unreachable due to choice synchronization. For example, $A\langle A\langle \texttt{Int}, \texttt{Bool}\rangle, \texttt{Int}\rangle \approx A\langle \texttt{Int}, \texttt{Int}\rangle$, since `Bool` is unreachable in $A\langle A\langle \texttt{Int}, \texttt{Bool}\rangle, \texttt{Int}\rangle$ because selection with either $A.1$ or $A.2$ yields `Int`. A corresponding relationship holds for $\lfloor M_2 \rfloor_{d.2}$.

The rule Cong defines that compatibility is a congruence relation. This rule allows us to replace a type M_1 in a context $M[]$ with a compatible type M_2. For example, since $\texttt{Bool} \approx B\langle \texttt{Bool}, \texttt{Bool}\rangle$, we have $A\langle \texttt{Int}, \texttt{Bool}\rangle \approx A\langle \texttt{Int}, B\langle \texttt{Bool}, \texttt{Bool}\rangle\rangle$ if we view $A\langle \texttt{Int}, [\,]\rangle$ as the context. Finally, the rule DynIntro states that if two types are compatible, replacing part of one type with `Dyn` preserves compatibility. This rule holds because `Dyn` is compatible with anything. By choosing M to be an empty context, this rule encodes $M \approx \texttt{Dyn}$ and thus $\texttt{Dyn} \approx M$ through T2.

To illustrate compatibility, we show $A\langle \texttt{Int}, \texttt{Dyn}\rangle \approx B\langle \texttt{Dyn}, \texttt{Int}\rangle$. This should hold, since both choice types only produce `Int` or `Dyn`, which are consistent with each other and themselves. We can start by $A\langle \texttt{Int}, \texttt{Int}\rangle \approx \texttt{Int}$ via T4 and $\texttt{Int} \approx B\langle \texttt{Int}, \texttt{Int}\rangle$ via T4 and T2. We can then use T3 to derive $A\langle \texttt{Int}, \texttt{Int}\rangle \approx B\langle \texttt{Int}, \texttt{Int}\rangle$. After that, we can apply DynIntro to replace the first `Int` in B with a `Dyn`, apply T2, and apply another DynIntro to replace the second `Int` in the choice A with a `Dyn`, yielding $B\langle \texttt{Dyn}, \texttt{Int}\rangle \approx A\langle \texttt{Int}, \texttt{Dyn}\rangle$. By applying T2 one more time, we can derive the original goal.

We demonstrate the correctness of $\approx$ by establishing its connection with type equivalence ($\equiv$) from [Chen et al. 2014] and type consistency ($\sim$) from [Siek and Taha 2006] through the following theorems. In the theorems we write $\lfloor M \rfloor_\delta \in V$ and $\lfloor M \rfloor_\delta \in G$ to denote that $\lfloor M \rfloor_\delta$ yields a variational type (no `Dyn`) and a gradual type (no variations), respectively. The first two theorems state the soundness of $\approx$; the third theorem states its completeness.

Theorem 4.1. *If* $M_1 \approx M_2$, *then* $\forall\delta.\lfloor M_1 \rfloor_\delta \in V \wedge \lfloor M_2 \rfloor_\delta \in V \Rightarrow \lfloor M_1 \rfloor_\delta \equiv \lfloor M_2 \rfloor_\delta$

Theorem 4.2. *If* $M_1 \approx M_2$, *then* $\forall\delta.\lfloor M_1 \rfloor_\delta \in G \wedge \lfloor M_2 \rfloor_\delta \in G \Rightarrow \lfloor M_1 \rfloor_\delta \sim \lfloor M_2 \rfloor_\delta$.

Theorem 4.3. $\forall\delta \lfloor M_1 \rfloor_\delta \equiv \lfloor M_2 \rfloor_\delta \vee \lfloor M_1 \rfloor_\delta \sim \lfloor M_2 \rfloor_\delta \Rightarrow M_1 \approx M_2$

Proof. All theorems can be proved by structural induction over these three relations. □

With $\approx$, we can formalize the application rule as follows.

$$\frac{\Gamma \vdash e_1 : M_1 \qquad \Gamma \vdash e_2 : M_2 \qquad dom\,(M_1) \approx M_2}{\Gamma \vdash e_1\ e_2 : cod\,(M_1)}$$

Based on this rule and $\approx$, we can calculate the type $B\langle \texttt{Dyn}, \texttt{Int}\rangle$ for `widthFunc 5`.

$$\pi ::= \bot \mid \top \mid d\langle \pi, \pi \rangle$$

$$\frac{\forall \delta . \lfloor \pi \rfloor_\delta = \top \Rightarrow \lfloor M_1 \rfloor_\delta \approx \lfloor M_2 \rfloor_\delta}{M_1 \approx_\pi M_2} \qquad \frac{\forall \delta . \lfloor \pi \rfloor_\delta = \top \Rightarrow \lfloor \Gamma \rfloor_\delta \vdash \lfloor e \rfloor_\delta : \lfloor M \rfloor_\delta}{\pi ; \Gamma \vdash e : M}$$

$$\frac{\forall \delta . \lfloor \pi \rfloor_\delta = \top \Rightarrow op\,(\lfloor M_1 \rfloor_\delta) \text{ is defined}}{op_\pi\,(M_1) \text{ is defined}} \qquad \frac{\forall \delta . \lfloor \pi \rfloor_\delta = \top \Rightarrow \lfloor M_1 \rfloor_\delta \; op \; \lfloor M_2 \rfloor_\delta \text{ is defined}}{M_1 \; op_\pi \; M_2 \text{ is defined}}$$

Fig. 5. Patterns and pattern-constrained relations and operations. *op* can be any unary or binary operation on types. The *is defined* stipulations in the premise mean that the operations are defined on their input types, as specified in Figure 2. The *is defined* in the conclusion indicates that the operation can be safely carried out on the migrational type when constricted by π.

4.3 Pattern-Constrained Judgments

The goal in this subsection is to type the application `widthFunc fixed` in `widthV`, thus solving challenge C2 for migrational typing. According to the type annotation of `widthV`, `widthFunc` has type $B\langle \texttt{Dyn}, \texttt{Int} \rightarrow \texttt{Int} \rangle$, and `fixed` has type $A\langle \texttt{Dyn}, \texttt{Bool} \rangle$. Since it is impossible to derive $B\langle \texttt{Dyn}, \texttt{Int} \rangle \approx A\langle \texttt{Dyn}, \texttt{Bool} \rangle$ (where the former is the domain of the function type and the latter is the type of the argument), the application rule from Section 4.2 fails to assign a type to `widthFunc fixed`. If we terminate the typing process, we will not be able to compute any type for `widthV`, failing to provide support for program migration.

While the compatibility check between $A\langle \texttt{Dyn}, \texttt{Int} \rangle$ and $B\langle \texttt{Dyn}, \texttt{Bool} \rangle$ fails, we observe that `Dyn`, the first alternative of A, is compatible with $B\langle \texttt{Dyn}, \texttt{Bool} \rangle$ and `Int`, the second alternative of A is compatible with `Dyn`, the first alternative of B. This suggests that we should describe compatibility at a more fine-grained level than simply saying whether two migrational types are compatible or not. We employ the idea of *typing pattern* (π) [Chen et al. 2012] to formalize this idea (see Figure 5). The patterns $\top$ and $\bot$ denote that the compatibility check succeeds and fails, respectively, and the choice pattern $d\langle \pi_1, \pi_2 \rangle$ describes the success or failure of compatibility checking within the context of choice d. We can now express the partial compatibility between $A\langle \texttt{Dyn}, \texttt{Int} \rangle$ and $B\langle \texttt{Dyn}, \texttt{Bool} \rangle$ by the typing pattern $A\langle \top, B\langle \top, \bot \rangle \rangle$.

In Figure 5 we define $M_1 \approx_\pi M_2$ such that M_1 and M_2 are compatible for all variants of π that are $\top$. In contrast, there is no requirement between M_1 and M_2 at other places. For example, $\texttt{Int} \approx_{A\langle \bot, \top \rangle} A\langle \texttt{Bool}, \texttt{Int} \rangle$, since `Int` $\approx$ `Int` at $A.2$ (and since we don't care that `Int` and `Bool` are incompatible at $A.1$).

The idea of constraining compatibility with patterns is quite powerful. We can even generalize it to typing judgments. Specifically, the typing relation $\pi ; \Gamma \vdash e : M$ holds if $\lfloor \Gamma \rfloor_\delta \vdash \lfloor e \rfloor_\delta : \lfloor M \rfloor_\delta$ for all δ such that $\lfloor \pi \rfloor_\delta = \top$. The advantage is that we don't need to worry about the typing in variants where π has $\bot$s. That also means that we should not use (or trust) the typing result at variants where π has $\bot$s. We formally define this relation in Figure 5. For example, since $\Gamma \vdash 1 : \texttt{Int}$ we have $A\langle \top, \bot \rangle ; \Gamma \vdash A\langle 1, \texttt{True} \rangle : \texttt{Int}$, even though `True` does not have the type `Int`. We can also generalize this idea to other operations, such as *dom* and *cod*, again defined in Figure 5.

Based on the idea of pattern-constrained judgments, we can define the following rule for typing function applications (where *dom* and *cod* will be formally defined in Figure 6):

$$\frac{\pi ; \Gamma \vdash e_1 : M_1 \qquad \pi ; \Gamma \vdash e_2 : M_2 \qquad dom_\pi(M_1) \approx_\pi M_2}{\pi ; \Gamma \vdash e_1 \; e_2 : cod_\pi(M_1)}$$

$\boxed{\pi; \Gamma \vdash e : M \mid \Omega}$

$$\textsc{Con}\ \frac{c \text{ is of type } \gamma}{\pi; \Gamma \vdash c : \gamma \mid \varnothing} \qquad \textsc{Var}\ \frac{x \mapsto M \in \Gamma}{\pi; \Gamma \vdash x : M \mid \varnothing}$$

$$\textsc{Abs}\ \frac{\pi; \Gamma, x \mapsto V \vdash e : M \mid \Omega}{\pi; \Gamma \vdash \lambda x.e : V \to M \mid \Omega} \qquad \textsc{AbsDyn}\ \frac{\pi; \Gamma, x \mapsto d\langle \mathtt{Dyn}, V\rangle \vdash e : M \mid \Omega \qquad d \textit{ fresh}}{\pi; \Gamma \vdash \lambda x : \mathtt{Dyn}.e : d\langle \mathtt{Dyn}, V\rangle \to M \mid \Omega \cup \{x \mapsto V\}}$$

$$\textsc{App}\ \frac{\pi; \Gamma \vdash e_1 : M_1 \mid \Omega_1 \qquad \pi; \Gamma \vdash e_2 : M_2 \mid \Omega_2 \qquad dom_\pi(M_1) \approx_\pi M_2}{\pi; \Gamma \vdash e_1\ e_2 : cod_\pi(M_1) \mid \Omega_1 \cup \Omega_2}$$

$$\textsc{If}\ \frac{(\pi; \Gamma \vdash e_j : M_j \mid \Omega_j)^{j:1..3} \qquad \mathtt{Bool} \approx_\pi M_1 \qquad M_2 \approx_\pi M_3}{\pi; \Gamma \vdash \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 : M_2 \sqcap_\pi M_3 \mid \Omega_1 \cup \Omega_2 \cup \Omega_3}$$

$$\textsc{Weaken}\ \frac{\pi; \Gamma \vdash e : M \mid \Omega \qquad \pi_1 \le \pi \qquad M =_{\pi_1} M_1}{\pi_1; \Gamma \vdash e : M_1 \mid \Omega}$$

$$\begin{aligned} dom\,(M_1 \to M_2) &= M_1 & cod\,(M_1 \to M_2) &= M_2 \\ dom\,(\mathtt{Dyn}) &= \mathtt{Dyn} & cod\,(\mathtt{Dyn}) &= \mathtt{Dyn} \\ dom\,(d\langle M_1, M_2\rangle) &= d\langle dom\,(M_1), dom\,(M_2)\rangle & cod\,(d\langle M_1, M_2\rangle) &= d\langle cod\,(M_1), cod\,(M_2)\rangle \end{aligned}$$

$$\begin{aligned} M \sqcap M &= M & M_{11} \to M_{12} \sqcap M_{21} \to M_{22} &= (M_{11} \sqcap M_{21}) \to (M_{12} \sqcap M_{22}) \\ \mathtt{Dyn} \sqcap M &= M & d\langle M_1, M_2\rangle \sqcap M &= d\langle M_1 \sqcap M, M_2 \sqcap M\rangle \\ M \sqcap \mathtt{Dyn} &= M & G \sqcap d\langle M_1, M_2\rangle &= d\langle G \sqcap M_1, G \sqcap M_2\rangle \end{aligned}$$

$$\pi \le \top \qquad \bot \le \pi \qquad \frac{\pi_1 \le \pi_2 \qquad \pi_2 \le \pi_3}{\pi_1 \le \pi_3} \qquad \frac{\pi_1 \le \pi_2}{\pi[\pi_1] \le \pi[\pi_2]} \qquad \frac{\pi_1 \approx \pi_2}{\pi_1 \le \pi_2}$$

Fig. 6. Typing rules. The operations *dom*, *cod*, and $\sqcap$ are undefined for cases that are not listed here. The operations dom_π, cod_π, and $\sqcap_\pi$ can be obtained from Figure 5.

With this new rule, which accounts for migrational types with type errors, we can revisit the problem of typing `widthFunc fixed`. Let $\pi = A\langle \top, B\langle \top, \bot\rangle\rangle$. Since `widthFunc` $\mapsto A\langle \mathtt{Dyn}, \mathtt{Int} \to \mathtt{Int}\rangle$ belongs to Γ, we have $\pi; \Gamma \vdash$ `widthFunc` $: M$, where $M = A\langle \mathtt{Dyn}, \mathtt{Int} \to \mathtt{Int}\rangle$. Similarly, we have $\pi; \Gamma \vdash$ `fixed` $: B\langle \mathtt{Dyn}, \mathtt{Bool}\rangle$. Next, $dom_\pi(M) = A\langle \mathtt{Dyn}, \mathtt{Int}\rangle$. As we have seen earlier, $A\langle \mathtt{Dyn}, \mathtt{Int}\rangle \approx_\pi B\langle \mathtt{Dyn}, \mathtt{Bool}\rangle$. Thus, all the premises of the application rule are satisfied, and we can derive $\pi; \Gamma \vdash$ `widthFunc fixed` $: A\langle \mathtt{Dyn}, \mathtt{Int}\rangle$. Based on the result pattern, we should not trust the typing information at the variant $\{A.2, B.2\}$ since $\lfloor \pi \rfloor_{\{A.2, B.2\}} = \bot$.

While pattern-constrained judgments simplify the presentation, there is still the challenge of how to find appropriate patterns, which are inputs to the typing relation. However, the pattern is determined by the typing constraints among the subexpressions. For example, the type of the argument must match the argument type of the function. The reason we use $A\langle \top, B\langle \top, \bot\rangle\rangle$ in typing `widthFunc fixed` is that the application is ill typed at $\{A.2, B.2\}$. Therefore, in a language with type inference, the pattern will be computed during the inference process (Sections 6 and 7).

4.4 Typing Rules

The typing rules are shown in Figure 6. They are based on the compatibility relation (Section 4.2) and pattern-constrained judgments (Section 4.3). The typing judgment has the form $\pi; \Gamma \vdash e : M \mid \Omega$ and expresses that e has type M under environment Γ constrained by the pattern π. The mapping

Ω collects the types that will be assigned to parameters if their Dyns are removed. We assume that parameter names from different functions are uniquely identified in the domain of Ω. The goal of Ω is to connect the typing rules here with those from Figure 2. We discuss this aspect in more detail in Section 4.5 where we investigate the properties of the type system.

The rules for constants (Con) and variables (Var) are straightforward. They hold for arbitrary patterns π because constants and bound variables are always well typed. Moreover, since the types remain unchanged, Ω is always $\emptyset$. The rule Abs for an abstraction whose parameter is not annotated with Dyn is conventional. In rule AbsDyn for an abstraction whose parameter is annotated with Dyn, we assign the parameter a choice type where the first alternative is Dyn implying that the Dyn is kept and the second alternative can be any type for the body to be well typed. This change information is recorded by extending the Ω returned from typing the body of the abstraction.

The App rule for applications is similar to the one in Section 4.3 except that we must combine the variational statifiers from typing the two subexpressions. The rule If types conditionals; it relies on an extended version of the meet operation ($\sqcap$) from Figure 2 that also handles choices.

The Weaken rule states that if a typing pattern can be used to derive a typing, then we can use a less-defined pattern to derive the same typing. The operation $=_{\pi_1}$ in the premise specifies that its arguments must be the same for places where π_1 has $\top$s. A typing pattern π_1 is *less defined* than π_2 if it contains $\bot$ values at least everywhere π_2 does. The purpose of Weaken is to make the typing process compositional. Without this rule, the whole typing derivation must use the same π. With this rule, we can use different patterns for typing the children of a construct but adjust them to use the same pattern when typing the construct itself.

The less-defined relation on patterns, written as $\pi_1 \leq \pi_2$, is formally defined in Figure 6. The first two rules defines that any pattern is less defined than $\top$ and more defined than $\bot$. The third rule defines that the relation is transitive. In the last two rules, we reuse the machineries defined for types to simplify the definition of the relation. The fourth rule states that the less-defined relation is a congruence. The fifth rule states that two compatible patterns satisfy the less-defined relation. Since a pattern cannot contain Dyn, $\pi_1 \approx \pi_2$ implies that π_1 and π_2 are equivalent.

4.5 Properties

This subsection investigates the properties of the type system. Specifically, we consider the relationship of the rules for migrational typing in Figure 6 and the original rules for gradual typing in Figure 2. We also consider the relation between different typing derivations $\pi; \Gamma \vdash e : M \mid \Omega$ when different πs and Ms are used for the same Γ and e, which addresses challenge C3 from Section 3.

We start by introducing some notation. We say a decision δ is *complete* for an expression e if it contains $d.1$ or $d.2$ for each d created while typing e. For π, a decision δ is complete if $\lfloor \pi \rfloor_\delta$ yields $\top$ or $\bot$. Note that a complete decision for π may not be complete for the expression since patterns compactly represent where typing succeeds and where it fails. For instance, while typing rowAtI, we created five choices A, B, D, E, and F for the dynamic parameters from left to right, respectively. Thus, each complete decision for rowAtI contains five selectors. One typing pattern for rowAtI is:

$$\pi_a = A\langle E\langle \top, \bot \rangle, B\langle E\langle \top, \bot \rangle, \bot \rangle \rangle$$

Both $\{A.1, E.1\}$ and $\{A.2, B.2\}$ are complete decisions for π_a but not for rowAtI. In the case that the whole migration space for an expression is well typed, then the pattern is simply $\top$ and the complete decision is $\{\}$. We use the notation $\delta|_2$ to collect all of choice names d such that $d.2 \in \delta$.

There is a close relation among δ, Ω (variational statifier), and ω (statifier). Specifically, during typing, for each dynamic parameter x, Ω includes a mapping $x \mapsto V$, where V is the type that will be assigned to the parameter once its Dyn annotation is removed. Therefore, given Ω and δ, we can

generate a statifier as follows, where $chc(x)$ returns the name of the choice created for x.

$$\Omega(\delta) = \{x \mapsto \lfloor V \rfloor_\delta \mid x \mapsto V \in \Omega \wedge chc(x) \in \delta|_2\}$$

For example, let

$$\Omega_a = \{\texttt{fixed} \mapsto \texttt{Bool}, \texttt{widthFunc} \mapsto \texttt{Int} \to \texttt{Int}\} \qquad \delta_a = \{A.2, B.1\}$$

then $\Omega_a(\delta_a) = \{\texttt{fixed} \mapsto \texttt{Bool}\}$.

The notation $G_1 \sqsubseteq G_2$ means that G_2 is more static than G_1; it is defined as follows.

$$T_1 \sqsubseteq T_2 \qquad \texttt{Dyn} \sqsubseteq G \qquad \frac{G_1 \sqsubseteq G_3 \qquad G_2 \sqsubseteq G_4}{G_1 \to G_2 \sqsubseteq G_3 \to G_4}$$

We further say that G_2 is better than G_1, written as $G_1 \preceq G_2$, if G_2 is strictly more static than G_1 or they are equally static but G_2 is more general than G_1. For example, $\texttt{Dyn} \to \alpha \preceq \texttt{Int} \to \texttt{Int}$ and $\texttt{Int} \to \texttt{Int} \preceq \texttt{Int} \to \alpha$.

We next demonstrate the correctness of our type system by showing that, at the places where the typing pattern is valid, it assigns the same types to all the programs in the migration space as the brute-force approach does.

Theorem 4.4 (Dyn removal soundess). *If* $\pi; \Gamma \vdash e : M \mid \Omega$, *then* $\forall \delta. \lfloor \pi \rfloor_\delta = \top \Rightarrow \Omega(\delta); \Gamma \vdash_{GC} e : \lfloor M \rfloor_\delta$.

This theorem states that, for any removal of Dyn annotations, the typing result encoded in migrational typing is the same as by typing the program with ITGL. For example, for $\pi'_a = A\langle \top, B\langle \top, \bot \rangle\rangle$ we get $\pi'_a; \Gamma \vdash \texttt{width} : M_a \mid \Omega_a$, where $M_a = A\langle \texttt{Dyn}, \texttt{Bool}\rangle \to B\langle \texttt{Dyn}, \texttt{Int} \to \texttt{Int}\rangle \to B\langle \texttt{Dyn}, \texttt{Int}\rangle$ and Ω_a is as defined earlier. We can verify $\Omega_a(\delta_a); \Gamma \vdash_{GC} \texttt{width} : \texttt{Bool} \to \texttt{Dyn} \to \texttt{Dyn}$ and $\lfloor M_a \rfloor_{\delta_a} = \texttt{Bool} \to \texttt{Dyn} \to \texttt{Dyn}$, where δ_a is as defined earlier.

Conversely, any removal of Dyn that yields a well-typed program is encoded in some typing derivation in migrational typing, as expressed in the following theorem.

Theorem 4.5 (Dyn removal completeness). *If* $\omega; \Gamma \vdash_{GC} e : G$, *then there exists some typing* $\pi; \Gamma \vdash e : M \mid \Omega$ *such that* $\lfloor M \rfloor_\delta = G$ *and* $\Omega(\delta) = \omega$ *for some* δ.

Theorem 4.4 can be proved by structural induction over the rules in Figure 6, and Theorem 4.5 can be proved by induction over the rules in Figure 2.

Next, we investigate the relation between different typings that can be derived for the same expression and environment. We observe that different typings can be combined to make the result as correct as possible (that is, to minimize $\bot$s in the result pattern) and as good as possible (that is, to make types more static and more general). Note that the typing process records all dynamic parameters and corresponding variational types in Ω. As a result, the domain of Ωs in different typings are the same.

Lemma 4.6. *If* $\pi_1; \Gamma \vdash e : M \mid \Omega$ *and* $\pi_2; \Gamma \vdash e : M \mid \Omega$, *then there is some typing* $\pi; \Gamma \vdash e : M \mid \Omega$ *such that* $\pi_1 \preceq \pi$ *and* $\pi_2 \preceq \pi$.

Lemma 4.7. *If* $\pi; \Gamma \vdash e : M_1 \mid \Omega_1$ *and* $\pi; \Gamma \vdash e : M_2 \mid \Omega_2$, *then there is some typing* $\pi; \Gamma \vdash e : M \mid \Omega$ *such that* $\forall \delta. \lfloor \pi \rfloor_\delta = \top \Rightarrow \lfloor M_1 \rfloor_\delta \preceq \lfloor M \rfloor_\delta \wedge \lfloor M_2 \rfloor_\delta \preceq \lfloor M \rfloor_\delta \wedge \Omega_1(\delta) \preceq \Omega(\delta) \wedge \Omega_2(\delta) \preceq \Omega(\delta)$.

In Lemma 4.7, we write $\omega_1 \preceq \omega_2$ if they share the same domain and for any x in the domain $\omega_1(x) \preceq \omega_2(x)$. The properties captured by the previous two lemmas can be combined to show that for any expression there exists a typing that has the most defined pattern and the most static and general result type. We refer to this typing as the most general static migrational typing, abbreviated as the *MGSM typing*.

Theorem 4.8 (MGSM Typing). *For any e and Γ, there is a MGSM typing $\pi; \Gamma \vdash e : M \mid \Omega$ such that for any $\pi_1; \Gamma \vdash e : M_1 \mid \Omega_1$, $\forall \delta. \lfloor \pi_1 \rfloor_\delta = \top \Rightarrow \lfloor \pi \rfloor_\delta = \top \wedge \lfloor M_1 \rfloor_\delta \preceq \lfloor M \rfloor_\delta$.*

Lemmas 4.6 and 4.7 and Theorem 4.8 can be proved by structural induction over the typing rules in Figure 6. To illustrate the use of Theorem 4.8, the MGSM typing for `width` is $\pi_b; \Gamma \vdash \texttt{width} : M_b \mid \Omega_b$, where

$$\Omega_b = \{\texttt{fixed} \mapsto \texttt{Bool}, \texttt{widthFunc} \mapsto \texttt{Int} \to \beta\} \qquad \pi_b = A\langle \top, B\langle \top, \bot \rangle \rangle$$
$$M_b = A\langle \texttt{Dyn}, \texttt{Bool} \rangle \to B\langle \texttt{Dyn}, \texttt{Int} \to \beta \rangle \to B\langle \texttt{Dyn}, \beta \rangle.$$

Theorem 4.8 implies that while an infinite number of typings may be derived (due to the $\bot$ pattern), we need only care about the MGSM typing since it encodes all the typings for the whole migration space. Sections 6 and 7 investigate the problem of computing the MGSM typing.

5 FINDING THE BEST MIGRATION

This section addresses challenge C4 from Section 3, that is, given the MGSM typing, how can we find the most static migrations? We address it by investigating the relationship between different migrations in Section 5.1 and developing an algorithm for extracting the most static migration from the typing pattern of a MGSM typing in Section 5.2.

We use the term *eliminator* to refer to complete decisions. We say that an eliminator δ_2 is *stricter* than an eliminator δ_1, written $\delta_1 \gg \delta_2$, if δ_2 does not select the left alternative (corresponding to `Dyn`) in more choices than δ_1. Formally,

$$\delta_1 \gg \delta_2 :\Leftrightarrow \forall d.d.1 \in \delta_2 \Rightarrow d.1 \in \delta_1$$

We say an eliminator δ is *valid* if $\lfloor \pi \rfloor_\delta = \top$ where π should be clear from the context. We will use δ^v to denote valid eliminators. For example, let

$$\delta_a^v = \{A.1, B.1\} \qquad \delta_b^v = \{A.1, B.2\} \qquad \delta_c^v = \{A.2, B.1\} \qquad \delta_d = \{A.2, B.2\}$$

then $\delta_a^v \gg \delta_b^v$ and $\delta_b^v \gg \delta_d$, but $\delta_b^v \not\gg \delta_c^v$. The eliminators δ_a^v, δ_b^v, and δ_c^v are valid, while δ_d is not, with respect to π_b from Section 4.5.

5.1 Relationships Between Migrations

Since every migration can be identified by an eliminator for the MGSM typing, and since stricter eliminators correspond to more static migrations, the problem of finding the most static migrations can be reduced to the problem of finding the strictest valid eliminators.

Instead of considering all valid eliminators for an expression (which is exponential in the number of dynamic parameters), we instead consider the valid eliminators of the typing pattern for the MGSM typing of the expression. The reason is that typing patterns are usually small, yielding fewer eliminators that we have to consider (in fact, later results will show that we don't have to consider even all of these). For example, the pattern π_a from Section 4.5 for `rowAtI` has only 5 eliminators while the expression itself has 32. As another example, from the pattern π_b, also from Section 4.5, we can see that $\delta_{ab}^v = \{A.1\}$ compactly represents δ_a^v and δ_b^v for `width`.

Our first question is whether any eliminator that is stricter than an invalid eliminator could be valid. This question seems irrelevant for this example because the invalid eliminator δ_d is already the strictest for π_b. However, this is not the case in general, and knowing the answer to this question helps us to prune the search space. For example, the eliminator $\{A.1, B.1, E.2\}$ is invalid for π_a, and we want to know whether any of the stricter eliminators—$\{A.1, B.2, E.2\}$, $\{A.2, B.1, E.2\}$, and $\{A.2, B.2, E.2\}$—are valid. The following theorem addresses our question.

Theorem 5.1 (Error Irrecoverability). *Let $\pi; \Gamma \vdash e : M \mid \Omega$ be an MGSM typing for e and Γ. If $\lfloor \pi \rfloor_\delta = \bot$, then $\forall \delta_1. \delta \gg \delta_1 \Rightarrow \lfloor \pi \rfloor_{\delta_1} = \bot$.*

This theorem implies that we can simply ignore invalid eliminators, and focus on valid ones, since all invalid eliminators lead to ill-typed expressions. The theorem can be proved by structural induction over the typing rules in Figure 6.

A valid eliminator for the typing pattern corresponds to potentially many valid eliminators for the expression. We say that a valid pattern eliminator δ_1 *covers* a valid expression eliminator δ_2 if $\delta_1 \subseteq \delta_2$. Among all the expression eliminators covered by a pattern eliminator, one is the strictest. For example, the eliminator δ^v_{ab} for pattern π_b covers the eliminators δ^v_a and δ^v_b for typing `width`, and δ^v_b is the strictest. As another example, the valid eliminator $\delta^v_{ae} = \{A.1, E.1\}$ for pattern π_a covers eight valid eliminators (two options for each of the three choice names that do not appear in the pattern) for typing `rowAtI`, and $\{A.1, E.1, B.2, D.2, F.2\}$ is the strictest among them.

Among all expression eliminators covered by a pattern eliminator, stricter ones yield better result types. This is expressed by the following theorem.

Theorem 5.2. *If $\pi; \Gamma \vdash e : M \mid \Omega$ is the MGSM typing for e and Γ, then $\delta^v_1 \gg \delta^v_2 \Rightarrow \lfloor M \rfloor_{\delta^v_1} \preceq \lfloor M \rfloor_{\delta^v_2}$.*

Based on Theorem 4.4, the result stated in Theorem 5.2 can be transformed into a property of the original ITGL type system in Figure 2. The theorem can thus be proved through a structural induction of the typing rules in Figure 2.

As an example illustrating Theorem 5.2, consider δ^v_a, δ^v_b, and M_b, introduced in Section 4.5. We can verify that both $\delta^v_a \gg \delta^v_b$ and $\lfloor M_b \rfloor_{\delta^v_a} \preceq \lfloor M_b \rfloor_{\delta^v_b}$, where $\lfloor M_b \rfloor_{\delta^v_a} = \texttt{Dyn} \to \texttt{Dyn} \to \texttt{Dyn}$, and $\lfloor M_b \rfloor_{\delta^v_b} = \texttt{Bool} \to \texttt{Dyn} \to \texttt{Dyn}$.

Theorem 5.2 provides a way to order the eliminators covered by a single pattern eliminator, but how about ordering different valid eliminators of the typing pattern? Considering pattern π_b, neither of the valid eliminators δ^v_b or δ^v_c is stricter than the other. Similarly, for pattern π_a, neither of the valid eliminators is stricter than the other. In fact, this property holds not only for these two examples, but also for a class of typing patterns that are in *pattern normal form*. We say a pattern is in normal form if it does not contain idempotent choices (choices with identical alternatives) and does not nest a choice in another choice with the same name (no dead alternatives). We capture this property in the following theorem.

Theorem 5.3 (Eliminator Incomparability). *Let $\pi; \Gamma \vdash e : M \mid \Omega$ be MGSM typing for e and Γ and π_1 be a normal form for π. Then for any δ^v_1 and δ^v_2 for π_1, $\delta^v_1 \not\gg \delta^v_2$, $\delta^v_2 \not\gg \delta^v_1$, and $\nexists \delta^v . \delta^v_1 \gg \delta^v \wedge \delta^v_2 \gg \delta^v$.*

Two eliminators that are incomparable with respect to $\gg$ will remove `Dyn`s for different parameters for the same expression, leading to types that are incomparable by $\sqsubseteq$ (defined in Section 4), and thus incomparable by $\preceq$. For example, since $\delta^v_b \not\gg \delta^v_c$ and $\delta^v_c \not\gg \delta^v_b$, we have $G_b \not\preceq G_c$ and $G_c \not\preceq G_b$, where $G_b = \lfloor M_b \rfloor_{\delta^v_b} = \texttt{Dyn} \to (\texttt{Int} \to \beta) \to \beta$ and $G_c = \lfloor M_b \rfloor_{\delta^v_c} = \texttt{Bool} \to \texttt{Dyn} \to \texttt{Dyn}$.

Combining Theorems 5.2 and 5.3, yields the following result about finding most static migrations. We develop an algorithm for extracting such migrations in Section 5.2.

Theorem 5.4 (Uniqueness of most static migrations). *Let $\pi; \Gamma \vdash e : M \mid \Omega$ be the MGSM typing for e and Γ, and π_1 be the normal form for π, then e has a unique most static migration if π_1 has only one valid eliminator. More generally, the number of most static migrations for e equals the number of valid eliminators for π_1.*

5.2 Extracting Most Static Migrations

The most static migrations for a program are identified by valid eliminators that describe whether to pick the `Dyn` annotation or the inferred type for each parameter. We compute this set of eliminators

from an MGSM typing in three steps: (1) simplify the typing pattern to its normal form, (2) collect the valid eliminators for the normal form, and (3) expand each valid eliminator into a strictest eliminator for the corresponding expression.

Simplifying a typing pattern to its normal form has two advantages. First, the valid eliminators are fewer and smaller. Second, we can use the result of Theorem 5.4 to find most static migrations. We use the following rules to simplify patterns to normal forms.

$$d\langle \pi, \pi \rangle \rightsquigarrow \pi \qquad d\langle \pi_1, \pi_2 \rangle \rightsquigarrow d\langle \lfloor \pi_1 \rfloor_{d.1}, \lfloor \pi_2 \rfloor_{d.2} \rangle \qquad \frac{\pi_1 \rightsquigarrow \pi_2}{\pi[\pi_1] \rightsquigarrow \pi[\pi_2]}$$

The first two rules remove idempotent choices and dead alternatives. The third rule enables simplifying parts of a larger pattern. For example, we can use the third and the first rule to simplify the pattern $\pi_c = A\langle E\langle B\langle \top, \top \rangle, \bot \rangle, B\langle E\langle \top, \bot \rangle, \bot \rangle \rangle$ to pattern π_a from Section 4.5.

We use the function $ve(\pi)$ to build the set of valid eliminators for a pattern π in normal form.

$$ve(\top) = \{\varnothing\} \qquad ve(\bot) = \varnothing \qquad ve(d\langle \pi_1, \pi_2 \rangle) = \{\{d.1\} \cup l \mid l \in ve(\pi_1)\} \cup \{\{d.2\} \cup r \mid r \in ve(\pi_2)\}$$

For example, $ve(\pi_a)$ yields $\{\delta_o^v, \delta_p^v\}$, where $\delta_o^v = \{A.1, E.1\}$ and $\delta_p^v = \{A.2, B.1, E.1\}$.

Finally, we use the following function $expand(\delta, \mathcal{D})$ to compute the strictest expression eliminator from the given pattern eliminator δ and the set $\mathcal{D}$ of all choice names in the expression.

$$expand(\delta, \mathcal{D}) = \delta \cup \{d.2 \mid d \in \mathcal{D} \wedge d.1 \notin \delta\}$$

For example, the set of choice names $\mathcal{D}$ for typing `rowAtI` is $\{A, B, D, E, F\}$, and $expand(\delta_o^v, \mathcal{D})$ yields $\{A.1, E.1, B.2, D.2, F.2\}$ and $expand(\delta_p^v, \mathcal{D})$ yields $\{A.2, B.1, E.1, D.2, F.2\}$.

Each expanded valid eliminator is a best eliminator that specifies how to migrate the program. For example, the first best eliminator for `rowAtI` above removes the `Dyn` annotation for `widthFunc`, `table`, and `i`, while the other best eliminator removes the `Dyn` annotation for `fixed`, `table`, and `i`.

Overall, these three steps provide a simple and efficient way to extract the most static migration from an MGSM typing. Usually the normal form of a typing pattern is small, and so has only a few valid eliminators. For example, if the program is still well typed after removing all `Dyn` annotations, then the pattern will be $\top$, which has only one valid eliminator (the empty set). Similarly, if the program is ill typed if any `Dyn` annotation is removed, then there is again just one valid eliminator.

Since normal forms are ideal, in Section 7, we will show how we can efficiently maintain patterns to be in normal form throughout the type inference process.

6 CONSTRAINT GENERATION

A subset of the constraint generation rules is presented in Figure 7. The judgment $\Gamma \vdash_C e : M \mid C \mid \pi$ states that under Γ, the expression e has type M when the constraint C is solved. Moreover, the type M is valid only for the variants described the $\top$ values of the typing pattern π. Accordingly, e and Γ are inputs, while π, M, and C are outputs. Note that we now omit the statifier Ω in constraint judgments since it is not needed for type inference. The syntax of constraints are as follows:

$$C ::= M_1 \approx^?_\pi M_2 \mid C \wedge C \mid d\langle C, C \rangle \mid \varepsilon$$

The first form represents type compatibility constraints. Often it is the case that two types are only partially compatible. The pattern π in the constraint allows this fact to be recorded when different constraints are combined. In some rules, the notation $\approx^?_\top$ is used to denote that a generated constraint will be solved successfully everywhere. The constraint $\alpha \approx^?_\top \kappa_1 \to \kappa_2$ that forces α to be a function type is such an example, where κ_1 and κ_2 are fresh type variables. The constraint $C_1 \wedge C_2$ defines the conjunction of two constraints C_1 and C_2, while the constraint $d\langle C_1, C_2 \rangle$ defines a choice

$$\textsc{ConC}\ \frac{c \text{ is of type } \gamma}{\Gamma \vdash_C c : \gamma \mid \varepsilon \mid \top} \qquad \textsc{VarC}\ \frac{x : M \in \Gamma}{\Gamma \vdash_C x : M \mid \varepsilon \mid \top} \qquad \textsc{AbsC}\ \frac{\Gamma \vdash_C e : M \mid C \mid \pi \qquad \alpha \text{ fresh}}{\Gamma \vdash_C \lambda x.e : \alpha \to M \mid C \mid \pi}$$

$$\textsc{AbsDynC}\ \frac{\Gamma, x \mapsto d\langle \mathtt{Dyn}, \alpha \rangle \vdash_C e : M \mid C \mid \pi \qquad \alpha \text{ fresh} \qquad d \text{ fresh}}{\Gamma \vdash_C \lambda x : \mathtt{Dyn}.e : d\langle \mathtt{Dyn}, \alpha \rangle \to M \mid C \mid \pi}$$

$$\textsc{DynAnnC}\ \frac{\Gamma \vdash_C e : M \mid C \mid \pi \qquad d \text{ fresh}}{\Gamma \vdash_C e :: \mathtt{Dyn} : d\langle \mathtt{Dyn}, M \rangle \mid C \mid \pi}$$

$$\textsc{AppC}\ \frac{\begin{array}{c}\Gamma \vdash_C e_1 : M_1 \mid C_1 \mid \pi_1 \qquad \Gamma \vdash_C e_2 : M_2 \mid C_2 \mid \pi_2 \qquad cod(M_1) \hookrightarrow (M_3, C_3, \pi_3) \\ dom(M_1, M_2) \hookrightarrow (C_4, \pi_4) \qquad \pi = \pi_1 \otimes \pi_2 \otimes \pi_3 \otimes \pi_4 \qquad C = C_1 \wedge C_2 \wedge C_3 \wedge C_4\end{array}}{\Gamma \vdash_C e_1\ e_2 : M_3 \mid C \mid \pi}$$

Fig. 7. Constraint generation rules.

$$\begin{array}{rl} dom\,(\mathtt{Dyn}, M) \hookrightarrow (\varepsilon, \top) & dom\,(\alpha, M) \hookrightarrow (\alpha \approx^?_\top \kappa_1 \to \kappa_2 \wedge \kappa_1 \approx^?_\pi M_2, \pi) \\ dom\,(M_{11} \to M_{12}, M) \hookrightarrow (M_{11} \approx^?_\pi M, \pi) & dom\,(d\langle M_1, M_2 \rangle, M) \hookrightarrow d\langle dom\,(M_1, M), dom\,(M_2, M) \rangle \\ dom\,(\_, \_) \hookrightarrow (\varepsilon, \bot) & \end{array}$$

$$\begin{array}{rl} cod\,(\mathtt{Dyn}) \hookrightarrow (\mathtt{Dyn}, \varepsilon, \top) & cod\,(\alpha) \hookrightarrow (\kappa_2, \alpha \approx^?_\top \kappa_1 \to \kappa_2, \pi) \\ cod\,(M_1 \to M_2) \hookrightarrow (M_2, \varepsilon, \top) & cod\,(d\langle M_1, M_2 \rangle) \hookrightarrow d\langle cod\,(M_1), cod\,(M_2) \rangle \\ cod\,(\_) \hookrightarrow (\kappa, \varepsilon, \bot) & \end{array}$$

Fig. 8. Auxiliary constraint generation functions. The notation $d\langle (C_1, \pi_1), (C_2, \pi_2) \rangle$ is expanded to $(d\langle C_1, C_2 \rangle, d\langle \pi_1, \pi_2 \rangle)$. Default cases are indicated by _ parameters and define behavior when no other case applies.

between two constraints. Finally, ε represents an empty constraint. This is needed to represent a judgment where no constraints are generated.

The rule AbsDynC generates constraints for abstractions with dynamic parameters. It helps facilitate migration by creating a fresh choice type with a left alternative containing Dyn and a right alternative containing a fresh type variable. The type variable is used to infer a new static type for the parameter, if possible. The rule AppC is more involved because both constraints and patterns from premises have to be combined. The typing pattern in the conclusion must be restricted enough to create a valid judgment but well defined enough to give useful information about where the judgment succeeds. The operation $\otimes$, defined below, can be viewed as a meet operation over the *less defined* partial order on typing patterns in Figure 6. It creates the greatest lower bound of two patterns, ensuring that the most defined pattern is used for an entire rule.

$$\top \otimes \pi = \pi \qquad \bot \otimes \pi = \bot \qquad d\langle \pi_1, \pi_2 \rangle \otimes d\langle \pi_3, \pi_4 \rangle = d\langle \pi_1 \otimes \pi_3, \pi_2 \otimes \pi_4 \rangle$$

$$d\langle \pi_1, \pi_2 \rangle \otimes \pi = d\langle \pi_1 \otimes \pi, \pi_2 \otimes \pi \rangle$$

We defer the rule for conditionals to the long version of this paper[2] since it can be derived systematically from the If rule in Figure 6, similarly as AppC is derived from App.

The rule AppC uses several auxiliary functions to generate constraints. These are defined in Figure 8 and take the form: $dom\,(M_1, M_2) \hookrightarrow (C, \pi)$ and $cod\,(M_1) \hookrightarrow (M_2, C, \pi)$, where $\hookrightarrow$ defines

[2] http://www.ucs.louisiana.edu/~sxc2311/ws/techreport/fullcgrules.pdf

a mapping from inputs to outputs. These functions implement the *dom* and *cod* operations defined for the declarative type system in Figure 6.

We illustrate *dom* by considering the example $dom\,(A\langle \texttt{Dyn}, \alpha\rangle, \texttt{Int})$. Since the first argument is a choice type, *dom* is recursively called on each alternative of A, yielding two subproblems $dom\,(\texttt{Dyn}, \texttt{Int})$ and $dom\,(\alpha, \texttt{Int})$. The first subproblem is handled by the case for `Dyn`, which returns $(\varepsilon, \top)$. The second subproblem is handled by the the case of *dom* for type variables. Since *dom* expects a function type, the constraint $\alpha \approx^?_\top \kappa_1 \to \kappa_2$ is generated. The argument type of the function is constrained with $\kappa_1 \approx^?_\pi \texttt{Int}$ to express that it must be compatible with `Int`. As a result, this subproblem returns $(\alpha \approx^?_\top \kappa_1 \to \kappa_2 \wedge \kappa_1 \approx^?_\pi \texttt{Int}, \pi)$, where π is created to collect the pattern from $\kappa_1 \approx^?_\pi \texttt{Int}$. The constraints and patterns for the subproblems are combined with the choice A, yielding the final constraint $A\langle \varepsilon, \kappa_1 \to \kappa_2 \wedge \kappa_1 \approx^?_\pi \texttt{Int}\rangle$ and pattern $A\langle \top, \pi\rangle$.

During constraint generation, a large pattern connecting symbolic patterns by $\otimes$ is generated. These symbolic patterns are placeholders that will be updated once the corresponding constraints are solved. For example, if the pattern $\pi_1 \otimes \pi_2$ and the constraints $\texttt{Dyn} \approx^?_{\pi_1} \texttt{Bool} \wedge A\langle \texttt{Int}, \texttt{Bool}\rangle \approx^?_{\pi_2} \texttt{Int}$ are generated, then the pattern will be updated to $\top \otimes \pi_2$ once the first constraint is solved. This update occurs because `Dyn` and `Bool` are compatible so π_1 will be updated to $\top$.

The following soundness and completeness theorems state that the constraint generation rules correspond to the declarative typing rules presented in Figure 6. In particular, Theorem 6.2 implies that constraint generation finds the MGSM typing. Following the spirit of Vytiniotis et al. [2011], we use the idea of sound and most-general solutions (θ) for constraints (C) in the following theorems (Vytiniotis et al. [2011] used the term *guess-free*). In Section 7, we provide a unification algorithm that generates solutions with these desired properties.

Theorem 6.1 (Soundness of Constraint Generation). *If* $\Gamma \vdash_C e : M \mid C \mid \pi$, *then* $\pi; \theta(\Gamma) \vdash e : \theta(M) \mid \Omega$ *for some* Ω, *where* θ *is a sound and most-general solution for* C.

Theorem 6.2 (Completeness of Constraint Generation). *If* $\pi_1; \theta_1(\Gamma) \vdash e : M_1 \mid \Omega$ *then* $\Gamma \vdash_C e : M \mid C \mid \pi$ *such that* $\pi_1 \leq \pi$, $\forall \delta. \lfloor \pi_1 \rfloor_\delta = \top \Rightarrow \lfloor \pi \rfloor_\delta = \top \wedge \lfloor M_1 \rfloor_\delta \preceq \lfloor \theta(M) \rfloor_\delta$, *and* $\theta_1 = \theta' \circ \theta$ *form some* θ', *where* θ *is a sound and most-general solution for* C.

Both theorems can be proved by structural induction over the rules in Figures 6 and 7. Intuitively, the rules in Figure 7 don't forget any constraints and no extra constraints are generated with respect to the declarative type system Figure 6.

7 UNIFICATION

This section presents a unification algorithm for solving the constraints generated in Section 6, thus completing the roadmap presented in Section 3.

7.1 Solving Compatibility Constraints

We first motivate the structure and design of the algorithm by using the following examples.

(i) $\alpha \approx^?_\pi \texttt{Dyn} \to \texttt{Int}$
(ii) $A\langle \texttt{Dyn}, \texttt{Bool}\rangle \approx^?_\pi \texttt{Int}$

Our solver must adhere to certain rules to ensure the correctness of type inference, including:

(I) `Dyn` is compatible with any type (Section 2.1).
(II) Type variables only substitute for static types (Section 4).
(III) The typing pattern produced must be as defined as possible (Section 4).

Problem (i) helps illustrate rule (II). Intuitively, α should substitute to a function type whose codomain is `Int`, but what should the domain be? Essentially, the domain should be an unconstrained type variable so that it can unify with a static type later, if necessary. As a result, we generate

$\mathcal{U} : C \to \theta \times \pi$
(a) $\mathcal{U}(\texttt{Dyn} \approx^?_\pi M) = (\emptyset, \top)$
(a*) $\mathcal{U}(M \approx^?_\pi \texttt{Dyn}) = \mathcal{U}(\texttt{Dyn} \approx^?_\pi M)$
(b) $\mathcal{U}(\alpha \approx^?_\pi M)$
$\mid \alpha \notin vars(M) \wedge \neg hasDyn(M) = (\{\alpha \mapsto M, \top)\}$
$\mid d \in choices(M) = \mathcal{U}(d\langle \alpha, \alpha \rangle \approx^?_\pi M)$
$\mid \alpha \notin vars(M) \wedge M$ is of form $M_1 \to M_2 =$
let $(\theta_1, \pi_1) = \mathcal{U}(\alpha \approx^?_\top \kappa_1 \to \kappa_2)$; $(\theta_2, \pi_2) = \mathcal{U}(\kappa_1 \to \kappa_2 \approx^?_{\pi_2} M_1 \to M_2)$ in $(\theta_2 \circ \theta_1, \pi_2 \otimes \pi_1)$
$\mid$ otherwise $= (\emptyset, \bot)$
(b*) $\mathcal{U}(M \approx^?_\pi \alpha) = \mathcal{U}(\alpha \approx^?_\pi M)$
(c) $\mathcal{U}(d\langle M_1, M_2 \rangle \approx^?_\pi d\langle M_3, M_4 \rangle) =$
let $(\theta_1, \pi_1) = \mathcal{U}(M_1 \approx^?_{\pi_1} M_3)$; $(\theta_2, \pi_2) = \mathcal{U}(M_2 \approx^?_{\pi_2} M_4)$; $\theta' = merge(d, \theta_1, \theta_2)$
in $(\theta', d\langle \pi_1, \pi_2 \rangle)$
(d) $\mathcal{U}(d\langle M_1, M_2 \rangle \approx^?_\pi M) = \mathcal{U}(d\langle M_1, M_2 \rangle \approx^?_\pi d\langle \lfloor M \rfloor_{d.1}, \lfloor M \rfloor_{d.2} \rangle)$
(d*) $\mathcal{U}(M \approx^?_\pi d\langle M_1, M_2 \rangle) = \mathcal{U}(d\langle M_1, M_2 \rangle \approx^?_\pi M)$
(e) $\mathcal{U}(T_1 \approx^?_\pi T_2) =$ if $robinson(T_1, T_2) = \theta'$ then $(\theta', \top)$ else $(\emptyset, \bot)$
(f) $\mathcal{U}(M_{11} \to M_{12} \approx^?_\pi M_{21} \to M_{22}) =$
let $(\theta_1, \pi_1) = \mathcal{U}(M_{11} \approx^?_{\pi_1} M_{21})$; $(\theta_2, \pi_2) = \mathcal{U}(\theta_1(M_{12}) \approx^?_{\pi_2} \theta_1(M_{22}))$ in $(\theta_2 \circ \theta_1, \pi_1 \otimes \pi_2)$
(g) $\mathcal{U}(\varepsilon) = (\emptyset, \top)$
(h) $\mathcal{U}(d\langle C_1, C_2 \rangle) =$ let $(\theta_1, \pi_1) = \mathcal{U}(C_1)$; $(\theta_2, \pi_2) = \mathcal{U}(C_2)$; $\theta' = merge(d, \theta_1, \theta_2)$ in $(\theta', d\langle \pi_1, \pi_2 \rangle)$
(i) $\mathcal{U}(C_1 \wedge C_2) =$ let $(\theta_1, \pi_1) = \mathcal{U}(C_1)$; $(\theta_2, \pi_2) = \mathcal{U}(\theta_1(C_2))$ in $(\theta_2 \circ \theta_1, \pi_2 \otimes \pi_1)$

Fig. 9. A unification algorithm.

the substitutions $\{\kappa_2 \mapsto \texttt{Int}\} \circ \{\alpha \mapsto \kappa_1 \to \kappa_2\}$. Since κ_1 is a fresh type variable that is not mapped to anything, it is unconstrained. In contrast, κ_2 is mapped to `Int`. This substitution satisfies both rules (I) and (II).

Problem (ii) demonstrates the need for error tolerance in solving constraints. The natural way to solve a choice constraint is to decompose it into two constraints. Doing this on constraint (ii) yields two subconstraints, $\texttt{Dyn} \approx^?_{\pi_1} \texttt{Int}$ and $\texttt{Bool} \approx^?_{\pi_2} \texttt{Int}$, where $\pi = A\langle \pi_1, \pi_2 \rangle$. According to rule (I), the first constraint is solved successfully and π_1 is updated to $\top$. The second constraint, however, fails to solve, since `Bool` cannot be made compatible with `Int`, so we update π_2 to $\bot$. Consequently, we update π to $A\langle \top, \bot \rangle$ to reflect that constraint solving fails in $A.2$. Choosing instead $\bot$ for π would yield a consistent result but would violate rule (III).

7.2 A Unification Algorithm

Figure 9 presents a unification algorithm $\mathcal{U}$, which takes in a constraint and produces a substitution θ and a pattern π. As said in Section 6, constraint solving also updates the values of patterns that are used as placeholders. The figure uses the following helper functions. The function $choices(M)$ returns the set of choice names in M; $vars(M)$ returns the set of type variables in V. The predicate, $hasDyn(M)$, determines whether `Dyn` occurs anywhere in M. The function, *merge*, combines the substitutions from solving the subproblems of a choice constraint. For example, given d, $\theta_1 = \{\alpha \mapsto \texttt{Int}\}$, and $\theta_2 = \{\alpha \mapsto \texttt{Bool}\}$, we have $merge(d, \theta_1, \theta_2)(\alpha) = \{\alpha \mapsto d\langle \texttt{Int}, \texttt{Bool} \rangle\}$. Formally, the definition of *merge* (for each α in $\theta_1 \cup \theta_2$) is:

$$merge(d, \theta_1, \theta_2)(\alpha) = d\langle get(\alpha, \theta_1), get(\alpha, \theta_2) \rangle \text{ where } \alpha \in dom(\theta_1) \cup dom(\theta_2)$$

$$get(\alpha, \theta) = \begin{cases} M & \alpha \mapsto M \in \theta \\ \kappa & otherwise \end{cases}$$

We now briefly walk through each case of $\mathcal{U}$. Case *(a)* handles the trivial constraints involving Dyn. Such constraints are simply discarded without generating any mapping. We return $\top$ as the pattern since Dyn is compatible with any type. More importantly for $\alpha \approx^?_\pi$ Dyn, case *(a)* takes priority over *(b)*, ensuring that the substitution $\{\alpha \mapsto \texttt{Dyn}\}$ is not generated. Case *(b)* unifies a type variable α with a migrational type M. This case includes many subcases. First, if M does not contain Dyn and α does not occur in M, then α is directly mapped to M. For example, given $\alpha \approx^?_\pi A\langle \texttt{Int}, \texttt{Bool}\rangle$, the substitution $\{\alpha \mapsto A\langle \texttt{Int}, \texttt{Bool}\rangle\}$ is returned and the π is updated to $\top$. Second, if M contains variation, the result is computed via case *(d)*. For example, the problem $\alpha \approx^?_\pi A\langle \texttt{Dyn}, \texttt{Int}\rangle$ is transformed into $A\langle \alpha, \alpha\rangle \approx^?_\pi A\langle \texttt{Dyn}, \texttt{Int}\rangle$. Next, if M is a function type that contains Dyn and α does not occur in M, then we transform α into a function type by using fresh type variables and delegate the solving to case *(f)*. The problem (i) in Section 7.1 falls in this case. If all previous cases fail, $\bot$ is returned, indicating that the constraint failed to solve.

Case *(c)* handles constraints involving two choice types that share an outer choice name. It decomposes the constraint into two smaller problems and solves them individually. For instance, consider the constraint $A\langle \texttt{Dyn}, \alpha\rangle \approx^?_\pi A\langle \texttt{Int}, \texttt{Bool}\rangle$. This constraint will be decomposed into $\texttt{Dyn} \approx^?_{\pi_1}$ Int and $\alpha \approx^?_{\pi_2}$ Bool, which will be solved by *(a)* and *(b)*, respectively. Case *(d)* unifies a choice type with another type not handled by case *(c)*. This case is reduced to *(c)* but it turns the RHS of the constraint into a choice type that shares the outer choice name with the LHS. One such example is $A\langle \texttt{Dyn}, \texttt{Int}\rangle \approx^?_\pi \texttt{Int}$ transforming into $A\langle \texttt{Dyn}, \texttt{Int}\rangle \approx^?_\pi A\langle \texttt{Int}, \texttt{Int}\rangle$. Case *(e)* unifies two static types and is delegated to the traditional Robinson's algorithm. Case *(f)* unifies two function types by unifying their respective argument and return types. Cases *(g)*, *(h)*, and *(i)* deal with non-compatibility constraints.

To keep patterns in normal form, we also perform the following optimizations to prevent idempotent choices patterns from being created. In cases *(c)* and *(f)* of $\mathcal{U}$, when creating the choice pattern $d\langle \pi_1, \pi_2\rangle$, we check if π_1 and π_2 are the same; if so, the choice pattern is replaced by π_1. In the last two cases of $\otimes$ in Section 6, we perform the same optimization. After this, the algorithm maintains patterns in normal forms since additionally the generated constraints do not contain dead alternatives and the case *(d)* of $\mathcal{U}$ prevents dead alternatives from being introduced.

7.3 Properties

We now investigate the properties of $\mathcal{U}$. First, $\mathcal{U}$ is terminating.

Theorem 7.1 (Termination). *Given C, $\mathcal{U}(C)$ terminates.*

Next, we show that $\mathcal{U}$ is correct by showing that it is both sound and complete. For simplicity, we state the result for constraints of the form $M_1 \approx^?_\pi M_2$ only. In fact, we can transform other forms into this form. For example, $d\langle M_{11} \approx^?_{\pi_1} M_{12}, M_{21} \approx^?_{\pi_2} M_{22}\rangle$ can be transformed into $d\langle M_{11}, M_{21}\rangle \approx^?_{d\langle \pi_1, \pi_2\rangle} d\langle M_{12}, M_{22}\rangle$. Note that π in the constraint is just a placeholder and will be updated when the constraint solving finishes.

Theorem 7.2 (Soundness). *If $\mathcal{U}(M_1 \approx^?_\pi M_2) = (\theta, \pi')$, then $\theta(M_1) \approx_{\pi'} \theta(M_2)$.*

Theorem 7.3 (Completeness). *Given $M_1 \approx^?_\pi M_2$, if $\theta_1(M_1) \approx_{\pi_1} \theta_1(M_2)$, then $\mathcal{U}(M_1 \approx^?_\pi M_2) = (\theta_2, \pi_2)$ such that $\pi_1 \leq \pi_2$ and $\theta_1 = \theta \circ \theta_2$ for some θ.*

All theorems can be proved by going through the cases of $\mathcal{U}$ in Figure 9.

8 EXTENSIONS

In the previous sections we focused on making a gradually typed program as static as possible while preserving type correctness. This corresponds to Q1 from Section 1.1. In Section 8.1, we

describe how Q2 and Q3 can be addressed with small changes to our approach. Then, in Section 8.2, we consider how to support additional language features in our migrational type system.

8.1 Flexible Migration of Gradual Programs

Question Q2 from Section 1.1 asks whether we can take user considerations into account when migrating gradual programs. For example, the user should be able to indicate that a particular parameter should remain dynamic even if it could be made static, or that a particular parameter must be made static. This can be supported within our framework by using the user's preferences to override the inferred typing pattern before converting that typing pattern into a best migration using the method in Section 5.2. Given an inferred pattern π, we override the typing pattern with $d_1\langle\pi, \bot\rangle$ to force the parameter associated with choice name d_1 to be dynamic and override the pattern with $d_2\langle\bot, \pi\rangle$ to force the parameter associated with d_2 to be static. Essentially, we take the user's preference into account by considering the alternative to be ill typed. After doing this successively for each of the user's preferences, we can renormalize the pattern and use the method in Section 5.2 to compute the migration (i.e. determine which `Dyn` annotations to remove). Note that forcing a parameter to be static could make a migration fail when otherwise it would not.

Question Q3 asks whether we can migrate an ill-typed program to a type-correct program by adding as few `Dyn` annotations as possible. This can be supported in our framework by treating both abstraction forms uniformly and typing all abstractions using the rule AbsDyn from Figure 6. With this change, any parameter can be either dynamically or statically typed and our results from Sections 4 and 5 ensure that we can migrate to a type correct result that is as static as possible.

The idea of adding `Dyn` annotations to remove static type errors is closely related to the idea of deferring type errors by Vytiniotis et al. [2012]. However, our approach works at a more fine-grained level since their approach does not allow type errors related to function parameters to be deferred.

8.2 Other Language Features

Our version of ITGL, given in Figure 6, restricts parameters to be either unannotated or annotated by `Dyn`. The formulation of gradual typing by Garcia and Cimini [2015] allows arbitrary gradual type annotations on parameters, and also supports type ascription, that is, asserting by $e :: G$ that expression e has type G.

We can extend our type system to support arbitrary gradual type annotations as follows. Given an abstraction $\lambda x : G.e$, if $G = \texttt{Dyn}$ or G is fully static, type the abstraction as usual; if G is a complex type containing `Dyn` types, replace G by a choice whose first alternative is G and whose second alternative replaces all dynamic parts by arbitrary types. For example, if $G = \texttt{Int} \to \texttt{Dyn} \to \texttt{Dyn}$, then the type of the parameter is $d\langle\texttt{Int} \to \texttt{Dyn} \to \texttt{Dyn}, \texttt{Int} \to V_1 \to V_2\rangle$, where d is fresh. To generate the corresponding constraint (Section 6), we replace V_1 and V_2 by fresh type variables.

We can extend our type system to support type ascription with the following typing rule.

$$\frac{\pi; \Gamma \vdash e : M \mid \Omega \qquad G \approx_\pi V \qquad M \approx_\pi d\langle G, V\rangle}{\pi; \Gamma \vdash (e :: G) : d\langle G, V\rangle \mid \Omega \cup \{e \mapsto V\}}$$

The second premise ensures that the static parts of the ascribed type G are copied to the second alternative of the choice. The third premise ensures that the type of the expression M is compatible with the ascribed type and also a corresponding type V with all `Dyn` types removed. We can update the the structure of Ω to accommodate this rule by defining its domain to be program locations rather than parameter names. We use e here as shorthand for the location of e.

Finally, we can also add support for let-polymorphism. The approach is straightforward, but the notations become heavier. We use $\overline{\alpha}$ to denote a list of type variables and $\{\overline{\alpha \mapsto V}\}$ to denote a set that includes $\alpha_1 \mapsto V_1, \ldots, \alpha_n \mapsto V_n$. The function $vars(\cdot)$ returns the free type variables in its

ID	Size	# Func.	# Para.	# Chg.	# Best	Gradual	Brute	Migrational
1	7	1	5	2	2	$1.6e^{-3}$	$4.4e^{-2}$	$2.3e^{-3}$
2	17	8	9	5	11	$3.6e^{-3}$	1.6	$1.2e^{-2}$
3	24	9	14	5	3	$6.4e^{-3}$	79.5	$1.7e^{-2}$
4	126	8	10	10	17	$1.9e^{-2}$	20.1	$3.5e^{-2}$
5	237	86	139	20	9	$4.5e^{-2}$	–	0.1
6	2,110	420	576	5	7	0.38	–	0.77
7	2,110	420	576	100	743	0.39	–	0.75
8	8,460	2,750	2,946	50	83	2.7	–	4.5
$9^\star$	10,000	2,392	4,923	50	379	4.2	–	13.3
$10^\star$	20,000	7,630	9,364	100	894	12.0	–	24.3

$\star$: 100 programs of this size were generated. Results are the average of all programs.

Fig. 10. Running time (in seconds) of migrational typing on various programs. Times are measured on a ThinkPad with 2.4GHz i7-5500U 4-core processor and 8GB memory running GHC 8.0.2 on Ubuntu 16.04. Each time is an average of 10 runs. The symbol – indicates that typing timed out after 1,000 seconds.

argument. The typing rules are standard except that when typing variable references (Var) we can only instantiate type schemas with variational types (V) and not migrational types (M).

$$\textsc{Let}\ \frac{\pi;\Gamma \vdash e_1 : M_1 \mid \Omega_1 \qquad \overline{\alpha} = \mathit{vars}(M_1) - \mathit{vars}(\Gamma) \qquad \pi;\Gamma, x \mapsto \forall\overline{\alpha}.M \vdash e_2 : M_2 \mid \Omega_2}{\pi;\Gamma \vdash \textbf{let}\ x = e_1\ \textbf{in}\ e_2 : M_2 \mid \Omega_1 \cup \Omega_2} \qquad \textsc{Var}\ \frac{x \mapsto \forall\overline{\alpha}.M \in \Gamma}{\pi;\Gamma \vdash x : \{\overline{\alpha \mapsto V}\}(M) \mid \varnothing}$$

In support of all of these extensions, the other machinery of our approach, including constraint generation, unification, and extracting the most static migration, can be reused.

9 EVALUATION

This section evaluates the performance of migrational typing. For this purpose, we have implemented a prototype in Haskell. The prototype implements the techniques developed in this paper. Besides the features presented in Sections 4.1 and 8.2, the prototype also supports recursive functions and a built-in list type, which is needed to encode the examples described below.

We have created a suite of programs for performance evaluation since no public benchmarks exist in this domain. We first took 10 well-typed programs from the student program database [van Keeken 2006], whose sizes range from 17 LOC to 80 LOC (not including blank lines). We then randomly combined and duplicated these programs to create several programs of various sizes, from 17 to 20,000 LOC. The created programs are all well-typed by construction, so we seed errors in the programs by randomly applying between 2 and 100 changes in each. Each change replaces one leaf of the AST (a variable reference or constant) with another leaf. The generated programs are summarized in columns 2–5 of Figure 10, showing size in LOC, number of functions, number of dynamic parameters, and the number of leaves that were changed. We generated 8 programs of increasing size, then 100 programs each of 10,000 LOC and 20,000 LOC.

For each generated program, we compared the runtime of migrational typing with standard gradual typing and with a brute-force strategy for most static migration for the program, shown in the last three columns of the table. We also report the number of most static migrations in column "# Best", computed using the method in Section 5.2. The time for gradual typing can be considered a baseline—this is the time to simply type the given program. The time for the brute-force strategy represents a naive approach to migrational typing, generating 2^n variants of a program with n

dynamic parameters, and gradually typing all of them. We omit the time for computing the most static migrations from the figure because the time is always within 0.05 seconds.

From the figure, we observe that the brute force approach is exponentially slower than gradual typing, as expected, successfully typing only the first four programs. On the other hand, migrational typing scales linearly with the size of the program and exhibits only a 2–4 times overhead over gradual typing. The figure also shows that the number of changed locations and the number of most static migrations have a minor impact on the performance of the migrational typing process. For example, while the programs 6 and 7 have very different values for these two columns, the running times of migrational typing is almost the same.

It is interesting to note that the number of most static migrations seems to be independent of the number of changes made to the program. For example, program 5 changes more leaves but has fewer most static migrations than program 4. In fact, the kind and number of changes matters much more than the raw number. For example, scattered changes (vs. localized changes) and changes that directly affect types (e.g. changing `not` to `succ`) tend to create more most static migrations.

Many programs in our dataset have a large number of most static migrations. In practice, this would make migration difficult since the user has to somehow compare them. However, examining the results reveals that larger programs divide into clusters of interrelated functions, each with a relatively small number of candidate migrations. The high number of migrations for the whole program is caused by considering the product of possibilities for each cluster. For example, program 8 includes 3 clusters with 4, 4, and 5 most static migrations, respectively. The product of these three decisions accounts for almost all of its 83 most static migrations. In a real programming language, such clusters naturally correspond to modules, so migrating programs module-by-module is likely to provide a much better user experience. We can also imagine other strategies for coping with large numbers of potential migrations. Allowing the user to guide the migration process, as described in Section 8.1, is one possibility. Or we can imagine querying the potential migrations to, say, find the one that removes the largest number of `Dyn` annotations among all most static migrations.

10 RELATED WORK

10.1 Annotation Upgrading and Migratory Typing

Tansey and Tilevich [2008] studied the problem of automatically upgrading annotations (such as types and access modifiers in Java) in legacy applications in response to the upgrading of, for example, testing frameworks and libraries. This is similar to our work in that it tackles the problem of migrating programs to a new version by changing annotations in the program. Their methodology is quite different however, in that it needs two example programs illustrating how annotations change between framework versions, so that their inference rules can learn the changes made in the examples. In contrast our approach only needs to reason about how type annotations affect the typing of the program, so migrating annotations requires only information attainable through the type system. Moreover, the kind of migrations are orthogonal. Their goal is to upgrade an entire codebase automatically to use a new framework, which means that they have one endpoint. Migrational typing presents all of the ways a programmer might want to change the types of their program by adjusting `Dyn` annotations, meaning that there are multiple endpoints.

Migratory typing [Tobin-Hochstadt et al. 2017] provides another approach to migrating dynamically typed code to statically typed code by creating a statically-typed sister language that interfaces seamlessly with the dynamically-typed language. While migration in migratory typing is manual, migrational typing supports systematically typing the whole migration space and automatically finding the best migrations. Migratory typing defined ideal migration units as parts of a system that are small enough for easy migration but also large enough to be separately typed and to interface

with untyped code without excessive runtime type checks. The idea of migration units could be integrated into migrational typing as discussed at the end of Section 9.

10.2 Relation to Gradual Typing

Work on gradual typing can be broadly defined along three dimensions. The first investigates the integration of gradual typing with advanced typing features, such as objects [Siek and Taha 2007], ownership types [Sergey and Clarke 2012], refinement types [Jafery and Dunfield 2017; Lehmann and Tanter 2017], session types [Igarashi et al. 2017], and union and intersection types [Castagna and Lanvin 2017]. From this perspective, our type system studies the combination of variational types and gradual types. Gradual languages with type inference [Garcia and Cimini 2015; Rastogi et al. 2012; Siek and Vachharajani 2008] were a large influence on migrational typing. While ITGL was used as the basis for formalizing our type system, we expect that our approach can be extended to handle other features in this line of work. The reason is that the idea and manipulation of variation is orthogonal to other type system features. In particular, the idea of type compatibility in Section 4.2 and the handling of type errors in Section 4.3 can be easily extended.

The second dimension studies runtime error localization and performance issues with sound gradual typing. The blame calculus [Wadler and Findler 2009] adapts a contract system that can blame less precise parts of a program when cast errors occur in a gradually typed language. Ahmed et al. [2011, 2017] extended that work to further handle polymorphic types. Takikawa et al. [2016] showed that sound gradually typed languages suffer from performance issues as more interactions between static code and dynamic code leads to frequent value casts. Confined Gradual Typing [Allende et al. 2014] provides constructs to control the flow of values between static and dynamic code, mitigating performance issues and making gradual typing more predictable.

Gradual type inference with flow-based typing [Rastogi et al. 2012] has been explored to make programs in dynamic object-oriented languages more performant. Since our work is formalized on ITGL, our work inherits the relations between ITGL and the flow-based inference [Garcia and Cimini 2015]. Additionally, while flow-based inference ensures that inferred type annotations do not cause runtime errors, our current formalization does not have this property because the underlying ITGL does not have it. In contrast, while our approach finds the best way (according to many criteria, such as adding as many annotations as possible) to add annotations, the flow-based inference only considers one way of inferring types. Thus, it would be an interesting future direction to combine migrational typing and flow-based inference to combine their benefits.

The final dimension studies the production of gradual type systems from specifications of static type systems. For example, Garcia et al. [2016] presented a way to create gradual type systems from static ones using techniques from abstract interpretation. The Gradualizer [Cimini and Siek 2016, 2017] can produce a gradual type system and dynamic semantics for a statically-typed language given its formal semantics. It is thus interesting to investigate how these approaches interact with variations in the future. Siek et al. [2015] discussed the criteria for gradual typing. We employed the criteria of the underlying ITGL to prove Theorem 5.1.

10.3 Variational Typing

This work reuses much machinery from variational typing [Chen et al. 2012, 2014] to support reuse when typing the whole migration space. Thus, migrational typing can be viewed as an application of variational typing. Variational typing has been employed to improve type inference of generalized algebraic data types [Chen and Erwig 2016], which uses variation types to represent potentially many types for a single expression. Variational typing has also been used to improve error locating in functional programs using counter-factual typing (CFT) [Chen and Erwig 2014a,b]. Both migrational typing and CFT use variational types to efficiently explore a large number of

hypothetical situations. A technical difference between CFT and migrational typing is that CFT tries to find a minimal change that would make an ill-typed program type correct. In contrast, migrational typing tries to remove `Dyn` annotations from as many parameters as possible. The process of extracting the maximum change for migrational typing (as described in Section 5.2) is well defined while finding the minimum change in CFT has to rely on heuristics due to the nature of type error debugging. Another difference is that migrational typing considers the interaction between variational types and gradual types. The idea of using pattern-constrained judgments in Section 4.3 yields a declarative specification for handling type errors, while previous applications of variational typing have had to explicitly track the introduction and propagation of type errors.

Variational typing is defined in terms of the choice calculus [Erwig and Walkingshaw 2011]. Other applications of the choice calculus include the development of variational data structures [Meng et al. 2017; Smeltzer and Erwig 2017; Walkingshaw et al. 2014] to support variational program execution [Chen et al. 2016; Erwig and Walkingshaw 2013; Nguyen et al. 2014], and view-based editing of variational programs [Stănciulescu et al. 2016; Walkingshaw and Ostermann 2014].

11 CONCLUSION

We have presented migrational typing, a type system that allows programs in an implicitly typed gradual language to be assigned a new type based on the possible removals of dynamic type annotations in the original program. Migrational typing conceptually types the whole migration space, marking where type errors occur so that it can safely present the possible migrations for the program. We have shown that the system can infer the most static possible types that can be assigned to a program and that this process can be constrained according to user defined criteria. Moreover, the migrational type system is sound and complete with respect to removing dynamic annotations in ITGL, and its constraint generation and unification algorithms are sound and complete. We have also shown that this approach is scalable, performing nearly exponentially better than the brute force approach of generating and typing the migration space separately. Migrational typing solves an important unaddressed problem in gradual typing, namely having a safe and efficient way to move around in the possible dynamic-static typing space for a program.

In future work, we plan to investigate whether migrational typing can statically reason about the number of dynamic casts that will be generated by different points in the migration space so that we can pick the program with the fewest generated casts to minimize performance overhead.

ACKNOWLEDGMENTS

We would like to thank Aseem Rastogi and the anonymous reviewers for many helpful comments that improved the quality of this paper. This work is partially supported by the National Science Foundation under the grants IIS-1314384 and CCF-1717300, and by AFRL Contract FA8750-16-C-0044 (via Raytheon BBN Technologies) under the DARPA BRASS program.

REFERENCES

Amal Ahmed, Robert Bruce Findler, Jeremy G. Siek, and Philip Wadler. 2011. Blame for All. *SIGPLAN Not.* 46, 1 (Jan. 2011), 201–214. https://doi.org/10.1145/1925844.1926409

Amal Ahmed, Dustin Jamner, Jeremy G. Siek, and Philip Wadler. 2017. Theorems for Free for Free: Parametricity, with and Without Types. *Proc. ACM Program. Lang.* 1, ICFP, Article 39 (Aug. 2017), 28 pages. https://doi.org/10.1145/3110283

Esteban Allende, Johan Fabry, Ronald Garcia, and Éric Tanter. 2014. Confined Gradual Typing. *SIGPLAN Not.* 49, 10 (Oct. 2014), 251–270. https://doi.org/10.1145/2714064.2660222

Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. 2016. *Feature-Oriented Software Product Lines.* Springer.

Giuseppe Castagna and Victor Lanvin. 2017. Gradual Typing with Union and Intersection Types. *Proc. ACM Program. Lang.* 1, ICFP, Article 41 (Aug. 2017), 28 pages. https://doi.org/10.1145/3110285

Sheng Chen and Martin Erwig. 2014a. Counter-factual Typing for Debugging Type Errors. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '14)*. ACM, New York, NY, USA, 583–594. https://doi.org/10.1145/2535838.2535863

Sheng Chen and Martin Erwig. 2014b. Guided Type Debugging. In *Int. Symp. on Functional and Logic Programming (LNCS 8475)*. 35–51. https://doi.org/10.1007/978-3-319-07151-0_3

Sheng Chen and Martin Erwig. 2016. Principal Type Inference for GADTs. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '16)*. ACM, New York, NY, USA, 416–428. https://doi.org/10.1145/2837614.2837665

Sheng Chen, Martin Erwig, and Eric Walkingshaw. 2012. An Error-tolerant Type System for Variational Lambda Calculus. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming (ICFP '12)*. ACM, New York, NY, USA, 29–40. https://doi.org/10.1145/2364527.2364535

Sheng Chen, Martin Erwig, and Eric Walkingshaw. 2014. Extending Type Inference to Variational Programs. *ACM Trans. Program. Lang. Syst.* 36, 1, Article 1 (March 2014), 54 pages. https://doi.org/10.1145/2518190

Sheng Chen, Martin Erwig, and Eric Walkingshaw. 2016. A Calculus for Variational Programming. In *European Conf. on Object-Oriented Programming (ECOOP)*. 6:1–6:26.

Matteo Cimini and Jeremy G. Siek. 2016. The Gradualizer: A Methodology and Algorithm for Generating Gradual Type Systems. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '16)*. ACM, New York, NY, USA, 443–455. https://doi.org/10.1145/2837614.2837632

Matteo Cimini and Jeremy G. Siek. 2017. Automatically Generating the Dynamic Semantics of Gradually Typed Languages. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017)*. ACM, New York, NY, USA, 789–803. https://doi.org/10.1145/3009837.3009863

Martin Erwig and Eric Walkingshaw. 2011. The Choice Calculus: A Representation for Software Variation. *ACM Trans. Softw. Eng. Methodol.* 21, 1, Article 6 (Dec. 2011), 27 pages. https://doi.org/10.1145/2063239.2063245

Martin Erwig and Eric Walkingshaw. 2013. Variation Programming with the Choice Calculus. In *Generative and Transformational Techniques in Software Engineering (LNCS 7680)*. 55–99.

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 (POPL '15)*. ACM, New York, NY, USA, 303–315. https://doi.org/10.1145/2676726.2676992

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 (POPL '16)*. ACM, New York, NY, USA, 429–442. https://doi.org/10.1145/2837614.2837670

Atsushi Igarashi, Peter Thiemann, Vasco T. Vasconcelos, and Philip Wadler. 2017. Gradual Session Types. *Proc. ACM Program. Lang.* 1, ICFP, Article 38 (Aug. 2017), 28 pages. https://doi.org/10.1145/3110282

Khurram A. Jafery and Joshua Dunfield. 2017. Sums of Uncertainty: Refinements Go Gradual. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017)*. ACM, New York, NY, USA, 804–817. https://doi.org/10.1145/3009837.3009865

Nico Lehmann and Éric Tanter. 2017. Gradual Refinement Types. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017)*. ACM, New York, NY, USA, 775–788. https://doi.org/10.1145/3009837.3009856

Meng Meng, Jens Meinicke, Chu-Pan Wong, Eric Walkingshaw, and Christian Kästner. 2017. A Choice of Variational Stacks: Exploring Variational Data Structures. In *Int. Work. on Variability Modelling of Software-Intensive Systems (VaMoS)*. ACM, 28–35.

Hung Viet Nguyen, Christian Kästner, and Tien N Nguyen. 2014. Exploring Variability-Aware Execution for Testing Plugin-Based Web Applications. In *Int. Conf. on Software Engineering*. ACM, 907–918.

Aseem Rastogi, Avik Chaudhuri, and Basil Hosmer. 2012. The Ins and Outs of Gradual Type Inference. *SIGPLAN Not.* 47, 1 (Jan. 2012), 481–494. https://doi.org/10.1145/2103621.2103714

Ilya Sergey and Dave Clarke. 2012. Gradual Ownership Types. In *Proceedings of the 21st European Conference on Programming Languages and Systems (ESOP'12)*. Springer-Verlag, Berlin, Heidelberg, 579–599. https://doi.org/10.1007/978-3-642-28869-2_29

Jeremy Siek and Walid Taha. 2007. Gradual Typing for Objects. In *Proceedings of the 21st European Conference on ECOOP 2007: Object-Oriented Programming (ECOOP '07)*. Springer-Verlag, Berlin, Heidelberg, 2–27. https://doi.org/10.1007/978-3-540-73589-2_2

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 (DLS '08)*. ACM, New York, NY, USA, Article 7, 12 pages. https://doi.org/10.1145/1408681.1408688

Jeremy G Siek, Michael M Vitousek, Matteo Cimini, and John Tang Boyland. 2015. Refined criteria for gradual typing. In *LIPIcs-Leibniz International Proceedings in Informatics*, Vol. 32. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

Karl Smeltzer and Martin Erwig. 2017. Variational Lists: Comparisons and Design Guidelines. In *Int. Work. on Feature-Oriented Software Development (FOSD)*. ACM, 31–40.

Ştefan Stănciulescu, Thorsten Berger, Eric Walkingshaw, and Andrzej Wąsowski. 2016. Concepts, Operations, and Feasibility of a Projection-Based Variation Control System. In *IEEE Int. Conf. on Software Maintenance and Evolution (ICSME)*. IEEE, 323–333.

Asumu Takikawa, Daniel Feltey, Ben Greenman, Max S. New, Jan Vitek, and Matthias Felleisen. 2016. Is Sound Gradual Typing Dead?. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '16)*. ACM, New York, NY, USA, 456–468. https://doi.org/10.1145/2837614.2837630

Wesley Tansey and Eli Tilevich. 2008. Annotation Refactoring: Inferring Upgrade Transformations for Legacy Applications. In *Proceedings of the 23rd ACM SIGPLAN Conference on Object-oriented Programming Systems Languages and Applications (OOPSLA '08)*. ACM, New York, NY, USA, 295–312. https://doi.org/10.1145/1449764.1449788

Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. 2014. A Classification and Survey of Analysis Strategies for Software Product Lines. 47, 1 (6 2014), 6:1–6:45.

Sam Tobin-Hochstadt, Matthias Felleisen, Robert Findler, Matthew Flatt, Ben Greenman, Andrew M. Kent, Vincent St-Amour, T. Stephen Strickland, and Asumu Takikawa. 2017. Migratory Typing: Ten Years Later. In *2nd Summit on Advances in Programming Languages (SNAPL 2017) (Leibniz International Proceedings in Informatics (LIPIcs))*, Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi (Eds.), Vol. 71. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 17:1–17:17. https://doi.org/10.4230/LIPIcs.SNAPL.2017.17

Peter van Keeken. 2006. *Analyzing Helium Programs Obtained Through Logging–The process of mining novice Haskell programs.* Master's thesis. Department of Information and Computing Sciences, Utrecht University.

Dimitrios Vytiniotis, Simon Peyton Jones, and José Pedro Magalhães. 2012. Equality Proofs and Deferred Type Errors: A Compiler Pearl. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming (ICFP '12)*. ACM, New York, NY, USA, 341–352. https://doi.org/10.1145/2364527.2364554

Dimitrios Vytiniotis, Simon Peyton jones, Tom Schrijvers, and Martin Sulzmann. 2011. Outsidein(x) Modular Type Inference with Local Assumptions. *J. Funct. Program.* 21, 4-5 (Sept. 2011), 333–412. https://doi.org/10.1017/S0956796811000098

Philip Wadler and Robert Bruce Findler. 2009. Well-Typed Programs Can'T Be Blamed. In *Proceedings of the 18th European Symposium on Programming Languages and Systems: Held As Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009 (ESOP '09)*. Springer-Verlag, Berlin, Heidelberg, 1–16. https://doi.org/10.1007/978-3-642-00590-9_1

Eric Walkingshaw, Christian Kästner, Martin Erwig, Sven Apel, and Eric Bodden. 2014. Variational Data Structures: Exploring Tradeoffs in Computing with Variability. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (Onward! 2014)*. ACM, New York, NY, USA, 213–226. https://doi.org/10.1145/2661136.2661143

Eric Walkingshaw and Klaus Ostermann. 2014. Projectional Editing of Variational Software. In *ACM SIGPLAN Int. Conf. on Generative Programming: Concepts and Experiences (GPCE)*. ACM, 29–38.