Migrating Gradual Types

Abstract

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 replacing dynamic types with fully static types for 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.
Migrating Gradual Types

John Peter Campora III and Sheng Chen
University of Louisiana at Lafayette
Martin Erwig and Eric Walkingshaw
Oregon State University

1 Introduction

Gradual typing promises to combine the benefits of static and dynamic typing in a single language. In the original formulation by Siek & 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.

In gradual typing research, it is quite common to start with simply typed lambda calculus and extend it with annotations for dynamic types (Siek & Vachharajani, 2008; Rastogi et al., 2012; Garcia & Cimini, 2015). A function parameter can be annotated with *(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. This work, extending Campora et al. (2018a), investigates the problem of migrating gradual programs to be as static as possible without introducing type errors.

To fully realize the benefits of gradual typing, we need the ability to navigate along a program’s dynamic-static typing spectrum, in order to make it more static or more dynamic.
Migrating Gradual Types

when and where the respective strengths of each are desired. Answering the following three questions will help harness the full power of gradual typing.1

Q1. Can we make a gradually typed program as static as possible while maintaining its well-typedness to keep it executable?

Q2. Can we introduce as few dynamic 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?

Q3. Can we address the previous questions while keeping some user-indicated parts static or 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.

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. In the first part (up until Section 7), we develop machinery for addressing the question Q1.

We develop solutions for Questions Q2 and Q3 in Sections 8 and 9.3, respectively.

We illustrate the challenges regarding Q1 by considering the following program written in the calculus by Garcia & Cimini (2015) extended with Haskell functions and notations, where parameters annotated with ⋆ 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.

```haskell
rowAtI headOrFoot (fixed::⋆) (widthFunc::⋆) (table::⋆) (border::⋆) (i::⋆) =
  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 is 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 ⋆ annotations. Removing ⋆ annotations turns out to be much trickier than we may expect. First, if we remove all ⋆ annotations, then type inference fails for

1 This paper focuses on the problem that only type annotations are changed while program text remains the same as programs are migrated. Recent work on program migration by Migeed & Palsberg (2019) took a similar approach.
John Peter Campora III, Sheng Chen, Martin Erwig, and Eric Walkingshaw

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 `⋆` 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 & 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 `⋆`. 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 `⋆` 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 `⋆` annotations as a migration space. The act of moving from one potential program to another by changing types is known as a migration. We say a program in the migration space has a most static type if removing any `⋆` 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.

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. 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 `⋆` annotations can be removed. In this paper, we mainly consider the removal of `⋆` annotations to support migrating to a more statically typed program; that is, we make types more precise (Siek & Taha, 2006). However, in Section 8,

---

2. https://circleci.com/blog/why-were-no-longer-using-core-typed/
Migrating Gradual Types

we describe how a dual approach can be developed to support the addition of \(^*\) annotations (addressing Q2). Also, in Section 9, we describe how the approach can be extended to support further migration scenarios (addressing Q3). In this work, our development focuses on the ITGL calculus. We leave the migration problem in presence of other dynamic and static language features to future work.

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 \(\text{rowAtI}\) function from Section 1.1. To decide whether we can remove the \(^*\) annotation, we 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 typing of the second (\(\text{row}\)) 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 \(\text{row}\) 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 \(^*\) 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 & 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, even in 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 relationships between the resulting types. To illustrate these relationships, we list a subset of the migration space for the \(\text{rowAtI}\) 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 2 is well typed, we can deduce that program 3 is also well typed since they differ only in the \( \ast \) 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 \( \ast \) annotations from a set of parameters \( P \), then (1) removing \( \ast \) 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 \( \ast \) 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 \( \ast \) annotations does not provide the same ordering. For example, program 3 removes more \( \ast \) annotations than program 4, but program 4 has a more static type.

The third observation is that, if removing all \( \ast \) annotations for a set of parameters causes a type error, then removing the \( \ast \) 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.

1.3 Relations with Other Work in Program Migration

The work by Migeed & Palsberg (2019) also studied the problem of program migration. However, there are many significant difference between our work and theirs.

Differences in techniques There is a fundamental difference in finding the migrations in these two approaches. For a given program, their approach finds migrations in the following steps. First, it generates a set of programs where each program replaces a \( \ast \) in the current program with a \( \text{Int} \), \( \text{Bool} \), or \( \ast \rightarrow \ast \). Second, it uses the type checking algorithm from Garcia & Cimini (2015) to type check the each program from the set. If a program does not type check, then it is not a migration of the original program. Otherwise, it is a migration, and the whole migration process is continued from the current program. The two-step process stops when no more programs type check. After this process finishes, all programs that type check are considered as possible migrations of the original program.

Figure 2 left illustrates the migration process of Migeed & Palsberg (2019) for the expression \( \lambda x: \ast . x . x \). In the first step, three programs are generated, each replacing the \( \ast \) with a more precise type. The programs \( \lambda x: \text{Int} . x . x \) and \( \lambda x: \text{Bool} . x . x \) do not type check. Therefore, they are not migrations of \( \lambda x: \ast . x . x \). In contrast, the program \( \lambda x: \ast . x . x \)
Migrating Gradual Types

Fig. 2: Programs explored for searching possible migrations in Migeed & Palsberg (2019) (left) and this work (right). Programs in blue type check and those in red do not type check. The dashed lines in the left subfigure denote that an infinite number of programs were omitted from it.

Fig. 3: Programs explored for finding migrations for \texttt{rowAtI} in our approach. These programs (configurations) constitute the full migration lattice (Takikawa et al., 2016) for the program \texttt{rowAtI}. Each configuration is identified by a sequence of “+/-” signs, with “+” (“-”) indicates that the corresponding * is kept (removed). A configuration with strictly more “-”s is more precise. We present several lines relating program precision and omit most of them for clarity.

type checks and is a migration. Moreover, program migrations are searched starting from

\[ \lambda x: \star \rightarrow \star, x. x. \]

Putting aside variational typing, our approach can be viewed as generating all the programs that are obtained by removing all combinations of the *s in the program. After that, we use the type inference algorithm from Garcia & Cimini (2015) to check the type correctness and infer the type of each program. All programs that are type correct are migrations of the original programs. Figure 2 right shows all programs generated in our approach. Since there is only one * in the expression, there are only two possible expressions that we need to investigate for migrations: the original expression and the one that removes the *.

To give a more straight view about what the whole search space looks like, we present in Figure 3 all the programs that are generated for finding migrations for \texttt{rowAtI}. Since \texttt{rowAtI} contains five *s, the total number of programs we need to investigate is 32. The figure uses a sequence of five + or − characters to denote each generated program. If the \( i \)th character is a +, then the \( i \)th * is kept. Otherwise, it is removed.
As argued in Section 1.1, in general it is necessary to explore all the generated programs to find the programs that remove as many $\star$s as possible. Our main goal in this paper is to use variational typing to make the exploration process efficient.

In summary, the main technical difference is that while Migeed & Palsberg (2019) intertwine program generation and type checking to find migrations, our approach can be viewed as an efficient way of first generating all programs and then using type inference to find all migrations.

Differences in behaviors The differences in techniques lead to several significant behavioral differences in these two approaches, discussed below.

First, the migration space could be infinite in Migeed & Palsberg (2019) but it is always finite in our approach. The main reason is that in their approach if a program in the migration space type checks, then programs with more precise type annotations will be generated, which may be well typed, yielding more programs being generated. One such example is in Figure 2. Replacing the original $\star$ with $\star \rightarrow \star$ makes the expressions type checks, and replacing any $\star$ with $\star \rightarrow \star$ will also type check. This process may be repeated infinitely. In Figure 2, we use dashed lines to indicate such infiniteness.

Instead, our approach generates exactly $2^n$ programs, where $n$ is the number of $\star$s in the expression. For example, for the expression $\lambda x: \star. x x$, our approach generates two expressions (including the original one), as can be seen from Figure 2.

Second, as Migeed & Palsberg (2019) use type checking from Garcia & Cimini (2015) while our approach uses type inference from Garcia & Cimini (2015) and it is well-known that type inference is often incomplete, their approach can lead to more precise program migrations than ours for certain programs. For example, for the expression $\lambda x: \star. x x$, their approach will generate a program $\lambda x: \star \rightarrow \star. x x$. As this program type checks, it is a valid migration. However, in our approach, we will check the expression $\lambda x. x x$, obtained by removing the $\star$ from the expression. For this expression, type inference generates two constraints: $\beta = \beta_1 \rightarrow \beta_2$ and $\beta_1 \sim \beta$, where $\beta$, $\beta_1$, and $\beta_2$ are three type variables. The unification algorithm in Garcia & Cimini (2015) fails to solve these two constraints due to occurs check. Consequently, type inference fails for this expression. As our type inference is a variational version of the one in Garcia & Cimini (2015), we also fail to infer a type for $\lambda x. x x$. As a result, no improvement is possible in our approach for $\lambda x: \star. x x$. In Section 9.2, we present an extension to our approach that could infer more precise types, including finding a migration for the expression $\lambda x: \star. x x$.

Their work uses the term “maximal migration” to denote a migration that can not be made more precise (any such effort leads to ill-typed programs). For certain programs, no maximal migrations exist. The expression $\lambda x: \star. x x$ is one such example. The reason is that a $\star$ in any migration can be replaced by a $\star \rightarrow \star$, thus more precise, without making the program ill-typed. In our work, we use the term “most static migration” to refer to migrations where no more $\star$s could be removed and replaced with fully static types. For $\lambda x: \star. x x$, the most static migration is itself (our extension in Section 9.2 finds more static migrations). In our approach, most static migrations always exist because among a finite number of migrations we can always find migrations that remove most $\star$s. In case no $\star$s can be removed and replaced with fully static types, the original expression is considered as the most static migration. Maximal migrations and most static migrations may coincide.
Migrating Gradual Types

For example, the programs in Figure 3 that are in blue and in fourth column are maximal and most static migrations.

Third, while Migeed & Palsberg (2019) find maximal migrations by generating more precise programs and type checking them individually, we use variational typing to increase the efficiency of finding most static migrations. We have done a simple evaluation and find out that their approach has an exponential complexity. In particular, adding a parameter with $\star$ type essentially increases the running time by three times. For example, it takes about $4.7 \times 10^{-5}$ seconds to find the max migration for the expression $\lambda x:\star.\text{succ}(\text{succ} x)$, $1.5 \times 10^{-4}$ seconds for the expression $\lambda x:\star.\lambda y:\star.x+y$, 28.67 seconds for $\lambda x:\star.x_1:\star.x_2:\star.x_3:\star.x_4:\star.x_5:\star.x_6:\star.y:\star.y + \text{succ}(x_5 (x_6 + x_4 (\text{succ} x_3) (\text{succ} (x_2 (x_1 + x+y))))))$, and 93.8 seconds for $\lambda x:\star.x_1:\star.x_2:\star.x_3:\star.x_4:\star.x_5:\star.x_6:\star.y:\star.y + \text{succ}(x_5 (x_6 + x_4 (\text{succ} x_3) (\text{succ} (x_2 (x_1 + x+y))))))$. For these four expressions, our approach takes $4.1 \times 10^{-4}$, $5.9 \times 10^{-4}$, $1.7 \times 10^{-3}$, and $1.9 \times 10^{-3}$ seconds, respectively.

The timing result indicates that the idea of variational typing indeed improves efficiency. We present more comprehensive performance evaluation in Section 10.

1.4 Additions in the Journal Version and Contributions

This paper extends Campora et al. (2018a) with the following additions.

- In Section 1.3, we discuss in depth the relation between our work and the work by Migeed & Palsberg (2019).
- In Section 8, we present a solution to fixing static type errors by introducing as few dynamic types as possible (question Q2), a dual problem to removing as many as dynamic types (question Q1).
- In Section 9.2, we present an extension to our constraint solving algorithm that enables us to find more precise migrations that the approach in Campora et al. (2018a) was not able to.
- In addition to the migration questions Q1 and Q2, we consider many other migration scenarios, such as finding the migrations that migrate the greatest number of parameters. We present the approaches to support them in Section 9.3. These approaches reuse or slightly adapt the machinery for supporting Q1, which demonstrates the potential of our approach for developing more complex migration scenarios.
- In Section 10, we expand our evaluation by converting programs in Grift Kuhlen Schmidt et al. (2019) to our language and measure their performances.
- We updated related work to discuss the relation with the latest work on gradual typing, including Migeed & Palsberg (2019), Campora et al. (2018b), and Phipps Costin et al. (2021).

Overall, this paper makes the following contributions.

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 & Taha, 2006), variational types (Chen et al., 2014), and error-tolerant typing (Chen et al.,
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 & 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 & Taha (2006) defined gradual typing for a simply typed...
Migrating Gradual Types

Syntax:

Expressions  \( e ::= c \mid x \mid \lambda x.e \mid \lambda x : * e \mid e e \mid \text{if } e \text{ then } e \text{ else } e \)

Static types  \( T ::= \gamma \mid \alpha \mid T \rightarrow T \)

Gradual types  \( G ::= \gamma \mid \alpha \mid G \rightarrow G \mid * \)

Statifier  \( \omega ::= \emptyset \mid \omega, x : T \)

Type system:

\[
\begin{array}{ll}
\text{CON} & \omega; \Gamma \vdash \gamma \quad \text{c is of type } \gamma \\
\text{VAR} & \omega; \Gamma \vdash \gamma \\
\text{ABS} & \omega; \Gamma, x : T \vdash G_1 \quad \omega; \Gamma \vdash \lambda x : \gamma e : G_2 \\
\text{ABS DYN} & \omega; \Gamma, x : \omega \vdash \text{or}(\omega(x), \star) \vdash G_3 \\
\text{APP} & \omega_1; \Gamma \vdash G_1 \\
& \omega_2; \Gamma \vdash G_2 \\
& \omega_1 \cup \omega_2; \Gamma \vdash \text{dom}(G) \sim G' \\
\text{If} & \omega_1 \cup \omega_2 \cup \omega_3; \Gamma \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : G_2 \cap G_3 \\
\end{array}
\]

Gradual type consistency:

\[
\begin{array}{cccc}
\text{C1} & \text{C2} & \text{C3} & \text{C4} \\
G \sim G & G \sim * & * \sim G & G_{11} \sim G_{21} \\
G_{11} \rightarrow G_{12} \sim G_{21} \rightarrow G_{22} \\
\end{array}
\]

Auxiliary definitions:

\[
\begin{array}{ll}
dom(G_1 \rightarrow G_2) = G_1 & \star \cap G = G \\
dom(*) = * & G \cap \star = G \\
\text{cod}(G_1 \rightarrow G_2) = G_2 & G \cap G = G \\
\text{cod}(*) = * & G_{11} \rightarrow G_{12} \cap G_{21} \rightarrow G_{22} = (G_{11} \cap G_{21}) \rightarrow (G_{12} \cap G_{22}) \\
\end{array}
\]

Fig. 4: Syntax and type system of ITGL, an implicitly typed gradual language. The operations dom, cod, and \( \cap \) are undefined for cases that are not listed here.


In this paper, we consider the migration of programs in implicitly typed gradual languages. In Figure 4, we present the syntax and type system of one such language, ITGL, which is adapted from Garcia & Cimini (2015) and forms the basis for this work. In the syntax, \( c \) ranges over constant values, \( x \) over variables, \( \gamma \) over constant types, and \( \alpha \) over type variables. There are two cases for abstraction expressions, one where the parameter is annotated by \( * \) 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 4 differs from the original in Garcia & 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 do not consider let-polymorphism here. These restrictions are made to simplify our
formalization later, but we show in Section 9 how they can be lifted. Second, the typing rules are parameterized by a statifier, $\omega$, which is used in the full migrational type system later (Section 4). A statifier is a mapping that maps parameter names that have $\star$ annotations to static types, making an expression to have a more static type. The statifier specifies what static types to assign to parameters whose $\star$ annotations will be removed. For simplicity, we assume parameters have unique names. In the type system as defined in Figure 4, $\omega$ is always empty, corresponding to the type system in Garcia & Cimini (2015).

In the type system for ITGL in Figure 4, the typing rules for constants and variables are standard. There are two rules for abstractions, $A B S$ for unannotated parameters which must have static types, and $A B S D Y N$ for annotated parameters which may have dynamic types. In $A B S D Y N$, we use $or(\omega(x), \star)$ to return $\omega(x)$ if $x \in \text{dom}(\omega)$ or $\star$ otherwise. Therefore, if $\omega$ is empty, then $or(\omega(x), \star)$ will always be $\star$.

Note that a statifier maps parameters to fully static types only, as can be seen from the definition of $\omega$ in Figure 4. As such, mappings such as $x \mapsto \star \rightarrow \text{Int}$ or $y \mapsto \star \rightarrow \star$ do not belong to $\omega$. This follows the spirit of Garcia & Cimini (2015) that inferred types should be fully static. Consequently, we can not find an $\omega$ to make the expression $\lambda x: \star \rightarrow x \rightarrow \star$ well typed, even though the expression $\lambda x: \star \rightarrow x \rightarrow \star$ is.

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, $\text{succ}$, has static type $\text{Int} \rightarrow \text{Int}$, both of the following programs in our Haskell-like notation should be accepted by gradual typing.

\[
\begin{align*}
\text{inc} \ (\text{num} :: \star) &= \text{succ} \ \text{num} \\
\text{foo} \ (f :: \star) &= f \ \text{True}
\end{align*}
\]

The $\text{APP}$ rule accommodates this with the help of a consistency relation, $\sim$, 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 $\text{dom}$ and $\text{cod}$ return the domain and codomain of a function type, respectively, or $\star$ for a dynamic type (reflecting the fact that $\star$ is equivalent to $\star \rightarrow \star$).

The gradual type consistency relation is defined in Figure 4 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, $\text{Int} \rightarrow \text{Int} \sim \text{Int} \rightarrow \star$ but not $\text{Int} \rightarrow \text{Int} \sim \text{Bool} \rightarrow \star$, 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, $\sqcap$, on gradual types. Intuitively, meet chooses the more static of two base types when one is $\star$. 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 & 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.

\[ \text{vfun} = A\langle \text{succ}, \text{even} \rangle 1 \]

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

A decision is a set of selectors; we use \( \delta \) to range over decisions. For each choice \( d \), a decision contains only one or neither of \( d.1 \) and \( d.2 \). 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, \( \text{succ} \ 1 \) is a plain expression and a variant of the variational expression \( \text{vfun} \).

A variational expression may contain several choices. Choices with the same name are synchronized and independent otherwise. For example, the variational expression \( A\langle \text{succ}, \text{even} \rangle A\langle 2, 3 \rangle \) has two variants, \( \text{succ} \ 2 \) and \( \text{even} \ 3 \), obtained by the decisions \( \{ A.1 \} \) and \( \{ A.2 \} \), respectively. The program \( \text{succ} \ 3 \) cannot be obtained through selection and so is not a variant of this expression. On the other hand, the variational expression \( A\langle \text{succ}, \text{even} \rangle B\langle 2, 3 \rangle \) has four variants, and we can obtain the variant \( \text{succ} \ 3 \) 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. We define variational types in Figure 5. They include constant types (\( \gamma \)), such as \( \text{Int} \) and \( \text{Bool} \), type variables (\( \alpha \)), function types, and choices over two alternatives.

All concepts and operations on variational expressions carry over to variational types. For example, Figure 5 defines selections on types. Selecting constant types (and type variables) with any selector yield themselves. For a function type, selection is recursively applied on the parameter type and return type. Selecting a choice type (\( d\langle V_1, V_2 \rangle \)) with a
John Peter Campora III, Sheng Chen, Martin Erwig, and Eric Walkingshaw

\[ V := \gamma \mid \alpha \mid V \rightarrow V \mid d \langle V, V \rangle \]

\[ |\gamma| = \gamma \quad |\alpha| = \alpha \quad |V_1 \rightarrow V_2| = |V_1| \rightarrow |V_2| \quad |d \langle V_1, V_2 \rangle| = |V_1| \cdot |d| \]

\[ |d \langle V_1, V_2 \rangle|_{d, 2} = |V_2|_{d, 2} \quad |d \langle V_1, V_2 \rangle|_{d, i} = d \langle |V_1|_{d, i}, |V_2|_{d, i} \rangle \quad |V|_{\delta} = |V|_{\delta} \]

VT-REF \quad V \equiv V \\
VT-SYM \quad V_1 \equiv V_2 \\
VT-TRANS \quad V_1 \equiv V_2 \\
VT-DEADIM \quad d \langle V_1, V_2 \rangle \equiv d \langle |V_1|_{d, 1}, |V_2|_{d, 2} \rangle \\
VT-CHOICE \quad \frac{V_1 \equiv V_1' \quad V_2 \equiv V_2'}{d \langle V_1, V_2 \rangle \equiv d \langle V_1', V_2' \rangle} \\
VT-FUN \quad \frac{V_1 \equiv V_1' \quad V_2 \equiv V_2'}{V_1 \rightarrow V_2 \equiv V_1' \rightarrow V_2'}

Fig. 5: Variational types, selection, and type equivalence

A selector that has the same choice name \((d, i)\) will yield the \(i\)th alternative. The selection is recursively applied to the alternative to eliminate all choices with the same name. For example, if we do not recursively select, \(\langle A, \langle A, \langle \text{Int, Bool} \rangle, \text{Bool} \rangle \rangle, A.1\) yields \(\langle \text{Int, Bool} \rangle\) while \(\text{Int}\) is the expected result, which could be achieved by recursively selecting \(\langle \text{Int, Bool} \rangle\) with \(A.1\). Selecting a choice type \((d \langle V_1, V_2 \rangle)\) with a selector \((d_1, i)\) that has a different choice name will apply the selection to both alternatives. Finally, selecting a type with a decision \((s : \delta)\) is recursively defined as first selecting the type with \(s\) and then selecting the resulting type with the decision \(\delta\).

It is natural to assign variational types to variational expressions. For example, \(A \langle \text{succ, even} \rangle\) has type \(A \langle \text{Int } \rightarrow \text{Int, Int } \rightarrow \text{Bool} \rangle\). 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 \(A \langle \text{Int } \rightarrow \text{Int, Int } \rightarrow \text{Bool} \rangle\), which is not a function type directly, but both variants of \(vfun, \text{succ 1 and even 1}\), are well typed.

Typing applications is supported in variational typing through the definition of a type equivalence relation (Chen et al., 2014), which is presented in Figure 5. Essentially, type equivalence 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, \(A \langle \text{Int } \rightarrow \text{Int, Int } \rightarrow \text{Bool} \rangle\), \(A \langle \text{Int, Int} \rangle \rightarrow A \langle \text{Int, Bool} \rangle\), and \(\text{Int } \rightarrow A \langle \text{Int, Bool} \rangle\) are all equivalent because selecting from each of them with \(\{A.1\}\) yields the same type \(\text{Int } \rightarrow \text{Int}\) and selecting from each of them with \(\{A.2\}\) yields the same type \(\text{Int } \rightarrow \text{Bool}\). As a result, we can say that \(vfun\) has the type \(\text{Int } \rightarrow A \langle \text{Int, Bool} \rangle\), which is a function type with the argument type \(\text{Int}\) matching the type of 1. We can thus assign the type \(V_{vfun} = A \langle \text{Int, Bool} \rangle\) 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 \( \delta \).

For example, \( \lfloor \text{vfun} \rfloor _{A.1} \) yields \( \text{succ} 1 \), which has the type \( \text{Int} \), the same as \( \lfloor V \text{vfun} \rfloor _{A.1} \). 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 \( \ast \) annotation a choice type whose the first alternative is a \( \ast \) and whose second alternative is a static type (In Section 9.1, we deal with parameter types that are partially static, such as \( \text{Int} \rightarrow \ast \)).

Consider, for example, the following function \( \text{widthV} \) that represents the variationally typed version of the function \( \text{width} \) (also shown below) for computing the table width in \( \text{rowAtI} \).

\[
\text{width} \left( \text{fixed} :: \ast \right) \left( \text{widthFunc} :: \ast \right) = \text{if fixed then widthFunc fixed else widthFunc} 5
\]

\[
\text{widthV} \left( \text{fixed} :: A(\ast, \text{Bool}) \right) \left( \text{widthFunc} :: B(\ast, \text{Int} \rightarrow \text{Int}) \right) =
\text{if fixed then widthFunc fixed else widthFunc} 5
\]

The function \( \text{widthV} \) encodes all four possible migrations of \( \text{width} \). If \( V_{\text{widthV}} \) is the type of \( \text{widthV} \), then \( \lfloor V_{\text{widthV}} \rfloor _{A.1, B.1} \) is the type for \( \text{width} \) with no \( \ast \) annotations removed, \( \lfloor V_{\text{widthV}} \rfloor _{A.2, B.2} \) is the type that replaces \( \ast \) with \( \text{Bool} \) for \( \text{fixed} \) and keeps \( \ast \) for \( \text{widthFunc} \), \( \lfloor V_{\text{widthV}} \rfloor _{A.1, B.2} \) is the type that keeps \( \ast \) for \( \text{fixed} \) but replaces \( \ast \) with \( \text{Int} \rightarrow \text{Int} \) for \( \text{widthFunc} \), and \( \lfloor V_{\text{widthV}} \rfloor _{A.2, B.2} \) is the type that removes both \( \ast \) annotations.

In order to successfully employ variational typing to improve the performance of migrational typing, several technical challenges must be addressed. Figure 6 presents challenges and relevant theorems. The challenge C2 (error tolerance) does not have any theorems associated with it so we omit it from the figure.

C1. We refer to this challenge **type compatibility**. In the presence of dynamic and variational types, we need to combine the type equivalence relation between variational types (marked as \( V \equiv \) in Figure 6) and the consistency relation between gradual types (marked as \( G \sim \) in the figure), which we refer to as the **compatibility**
relation (marked as \( M \approx \) in the figure). After introducing the syntax of the migrational type system in Section 4.1, we address this problem in Section 4.2. Theorems 1 through 3 prove that the combination is correct.

C2. We refer to this challenge error tolerance. 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. We refer to this challenge best typing. In the brute force approach, we need to generate all expressions \((e_1, e_2, \ldots \text{ in Figure 6})\) from the given expression \((e \text{ in the figure})\) by removing all combinations of \( \star \). These expressions will need to be typed using the type system \( \vdash_{GC} \) introduced in Figure 4. Our type system (presented in Section 4.4) types the expression \( e \) directly once without generating other programs (the judgment \( \pi, \Gamma \vdash e : M | \Omega \) in Figure 6). We thus need to show that our type system, by typing only one expression, essentially types all possible expressions that could be generated. Theorems 4 and 5 prove that this is indeed the case.

In \( \text{mitm} \), 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. Theorem 6 in Section 4.5 proves 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. We refer to this challenge migration extraction. In brute force approach, we need to compare typing results for all generated expressions to determine the most static migrations. While we could type just the original expression once with the best migrational typing, we need to find out the most static migrations from the typing result. This may also require the comparison of an exponential number of result types for the migration space. Fortunately, Theorems 7 through 10 prove that an efficient algorithm exists for finding most static migrations. In Section 5.2, we develop such an algorithm.

C5. We refer to this challenge type inference. In challenge C3 (best typing) 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 (constraint generation \( \vdash_C \) in Figure 6) and 7 (constraint solving \( \vdash \) in Figure 6). Theorems 11 through 15 prove desired properties of type inference.

4 Migrational Type System

This section addresses the challenges C1 (type compatibility)–C3 (best typing) 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 (type compatibility). A pattern-constrained typing relation is introduced in Section 4.3 and defined via typing rules in Section 4.4, addressing C2 (error tolerance). Finally, the properties of this type system are discussed in Section 4.5, addressing C3 (best typing).
Migrating Gradual Types

4.1 Syntax

The syntax of expressions, types, and environments is given in Figure 7. The metavariables we use to range over the relevant symbol domains are listed at the top of the figure. For type variables, we typically use $\beta$ to denote the result type of a function application during constraint generation and $\kappa$ 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 $\star$, or a choice of restricted migrational types; however, the more general syntax defined in Figure 7 is needed during the typing process. In Section 9.1, we extend our framework to allow an arbitrary mix of $\star$ and static types for top-level parameters. We also define type context to facilitate our presentations of both the type system and proofs.

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 $\omega$ records one way of making a program more static (by removing some subset of $\star$ annotations). A variational statifier $\Omega$ instead compactly encodes all possible statifiers for an expression. Since we want migration in our formalization to assign static types to parameters whose $\star$ annotations are removed, $\Omega$ 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 $f \mapsto \alpha \to \alpha \to \alpha \to \alpha$ and $x \mapsto \star$ in $\Gamma$. Now, when typing the application $f \, x$, we will substitute $\{ \alpha \mapsto \star \}$, yielding $\star \to \star \to \star \to \star$ as the type of $f \, x$. However, this implies that $f \, x \, 2 \, \text{True}$ is well typed, even though this violates the initial static type of $f$. The idea of substituting type variables with variational types but not migrational types is reminiscent
John Peter Campora III, Sheng Chen, Martin Erwig, and Eric Walkingshaw

\[
\begin{align*}
\text{MT-REFL} & \quad M \approx M \\
\text{MT-SYM} & \quad M_1 \approx M_2 \quad M_2 \approx M_1 \\
\text{MT-VTRANS} & \quad V_1 \approx V_2 \quad V_2 \approx V_3 \quad V_1 \approx V_3 \\
\text{MT-IDEMP} & \quad d\langle M, M \rangle \approx M \\
\text{MT-DEADELM} & \quad d\langle M_1, M_2 \rangle \approx d\langle \langle M_1 \rangle_{d.1}, \langle M_2 \rangle_{d.2} \rangle \\
\text{MT-CONG} & \quad M_1 \approx M_2 \quad M[M_1] \approx M[M_2] \\
\text{MT-DNINTRO} & \quad M_1 \approx M_2 \quad M_1 \approx M_2[\alpha] 
\end{align*}
\]

Fig. 8: Rules defining type compatibility

of Guha et al. (2007), where only certain contracts could be used to instantiate parametric contract variables. 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 \texttt{widthV} example from 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 \texttt{fixed} and the application \texttt{widthFunc 5 in widthV}.

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

These two examples demonstrate that we need a notion of \textit{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 & Taha, 2006). The definition of compatibility \( (M_1 \approx M_2) \) is given in Figure 8. The relation is reflexive (MT-REFL) and symmetric (MT-SYM). The relation is transitive (MT-VTRANS) in the case that no *s are present, which we indicate by using the metavariable for variational types \((V)\).

The rules MT-IDEMP and MT-DEADELM specify compatibility under choice type simplification. Rule MT-IDEMP states that a choice with identical alternatives is compatible with its alternatives. Rule MT-DEADELM says that two types are compatible under elimination of dead alternatives. Note that the operation \([M_1]_{d.1}\) 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, \(A\langle\texttt{Int,Bool}\rangle,\texttt{Int}\) \(\approx A\langle\texttt{Int,Int}\rangle\), since \texttt{Bool} is unreachable in \(A\langle\texttt{Int,Bool}\rangle,\texttt{Int}\) because selection with either \texttt{A.1} or \texttt{A.2} yields \texttt{Int}. A corresponding relationship holds for \([M_2]_{d.2}\).

The rule MT-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} \(\approx B\langle\texttt{Bool,Bool}\rangle\), we have \(A\langle\texttt{Int,Bool}\rangle \approx A\langle\texttt{Int,B\langle\texttt{Bool,Bool}\rangle}\rangle\) if we view \(A\langle\texttt{Int,[]}\rangle\)
Migrating Gradual Types

as the context. Finally, the rule MT-DYNINTRO states that if two types are compatible, replacing part of one type with * preserves compatibility. This rule is correct because * is compatible with anything. By choosing M to be an empty context, this rule encodes \( M \approx \ast \) and thus \( \ast \approx M \) through MT-SYM.

To illustrate compatibility, we show \( A(\text{Int}, \ast) \approx B(\ast, \text{Int}) \). This should hold, since both choice types only produce \text{Int} or \ast, which are consistent with each other and themselves.

We can start by \( A(\text{Int}, \text{Int}) \approx \text{Int} \) via MT-IDEMP and \( \text{Int} \approx B(\text{Int}, \text{Int}) \) via MT-IDEMP and MT-SYM. We can then use MT-VTRANS to derive \( A(\text{Int}, \text{Int}) \approx B(\text{Int}, \text{Int}) \). After that, we can apply MT-DYNINTRO to replace the first \text{Int} in \( B \) with a \ast, apply MT-SYM, and apply another MT-DYNINTRO to replace the second \text{Int} in the choice \( A \) with a \ast, yielding \( B(\ast, \text{Int}) \approx A(\text{Int}, \ast) \). By applying MT-SYM one more time, we can derive the original goal.

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

\[
\Gamma \vdash e_1 : M_1 \quad \Gamma \vdash e_2 : M_2 \quad \text{dom}(M_1) \approx M_2
\]

\[
\Gamma \vdash e_1 e_2 : \text{cod}(M_1)
\]

Based on this rule and \( \approx \), we can calculate the type \( B(\ast, \text{Int}) \) for \text{widthFunc} 3.

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 & Taha (2006) through the following theorems. In the theorems we write \( [M]_\delta \in V \) and \( [M]_\delta \in G \) to denote that \( [M]_\delta \) yields a variational type (no \( \ast \)) and a gradual type (no variations), respectively. The first two theorems state the soundness of \( \approx \); the third theorem states its completeness.

**Theorem 1 (Compatibility encodes equivalence)**

If \( M_1 \approx M_2 \), then \( \forall \delta. [M_1]_\delta \in V \land [M_2]_\delta \in V \Rightarrow [M_1]_\delta \equiv [M_2]_\delta \)

**Theorem 2 (Compatibility encodes consistency)**

If \( M_1 \approx M_2 \), then \( \forall \delta. [M_1]_\delta \in G \land [M_2]_\delta \in G \Rightarrow [M_1]_\delta \sim [M_2]_\delta \).

**Theorem 3 (Equivalence and consistency imply compatibility)**

\( \forall \delta. [M_1]_\delta \equiv [M_2]_\delta \lor [M_1]_\delta \sim [M_2]_\delta \Rightarrow M_1 \approx M_2 \)

### 4.3 Pattern-Constrained Judgments

The goal in this subsection is to type the application \text{widthFunc} \_\text{fixed} in \text{widthV}, thus solving challenge C2 (error tolerance) for migrational typing. According to the type annotation of \text{widthV}, \text{widthFunc} has type \( B(\ast, \text{Int} \rightarrow \text{Int}) \), and \text{fixed} has type \( A(\ast, \text{Bool}) \).

Since it is impossible to derive \( B(\ast, \text{Int}) \approx A(\ast, \text{Bool}) \) (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 \text{widthFunc} \_\text{fixed}. If we terminate the typing process, we will not be able to compute any type for \text{widthV}, failing to provide support for program migration.

While the compatibility check between \( A(\ast, \text{Int}) \) and \( B(\ast, \text{Bool}) \) fails, we observe that \( \ast \), the first alternative of \( A \), is compatible with \( B(\ast, \text{Bool}) \) and \text{Int}, the second alternative of \( A \), is compatible with \( \ast \), the first alternative of \( B \). This suggests that we should


\[ \pi := \bot \mid T \mid d(\pi, \pi) \]

\[
\begin{align*}
[T]_\delta &= T \\
[\bot]_\delta &= \bot \\
\langle d(\pi_1, \pi_2) \rangle_{d, i} &= \langle [\pi_1]_{d, i}, [\pi_2]_{d, i} \rangle \\
\langle \pi \rangle_{(\pi, \delta)} &= \langle [\pi]_{\delta} \rangle
\end{align*}
\]

**PATCOMP**
\[ \forall \delta. [\pi]_\delta = T \Rightarrow M_1 \pi M_2 \]

**PATUNARY**
\[ \forall \delta. [\pi]_\delta = \top \Rightarrow op(M_1) \text{ is defined} \]

**PATBINARY**
\[ \forall \delta. [\pi]_\delta = \top \Rightarrow M_1 \text{ op } M_2 \text{ is defined} \]

Fig. 9: Patterns and pattern-constrained relations and operations. op can be any unary or binary operation on types. The \textit{is defined} stipulations in the premise mean that the operations are defined on their input types, as specified in Figure 4. The \textit{is defined} in the conclusion indicates that the operation can be safely carried out on the migrational type when constricted by \pi.

describe compatibility at a more fine-grained level than simply saying whether or not two migrational types are compatible. We employ the idea of \textit{typing patterns} (\pi) (Chen et al., 2012) to formalize this idea (see Figure 9). The patterns \top and \bot denote that the compatibility check succeeds and fails, respectively, and the choice pattern \langle \pi_1, \pi_2 \rangle describes the success or failure of compatibility checking within the context of choice \pi.

In Figure 9, we also define selection on patterns, which is similar to selection on types ([V]_\delta) in Figure 5. On page 13, we gave a detailed explanation on selection on types, and we skip the explanation of selection on patterns here.

We can now express the partial compatibility between \(A\langle *, \text{Int} \rangle\) and \(B\langle *, \text{Bool} \rangle\) by the typing pattern \(A\langle \top, B(\top, \bot) \rangle\). It is also possible to give some pattern that has an identical effect, such as the pattern \(B\langle \top, A(\top, \bot) \rangle\).

In Figure 9 we define \(M_1 \approx_\pi M_2\) such that \(M_1\) and \(M_2\) are compatible for all variants of \(\pi\) that are \(\top\). In contrast, there is no requirement between \(M_1\) and \(M_2\) at other places. For example, \(\text{Int} \approx_{A(\bot, \top)} A(\text{Bool}, \text{Int})\), since \(\text{Int} \approx \text{Int}\) at A.2 (and since we do not care that \(\text{Int}\) and \(\text{Bool}\) 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 \(\pi; T \vdash e : M\) holds if \([\Gamma]_\delta \vdash [e]_\delta : [M]_\delta\) for all \(\delta\) such that \([\pi]_\delta = \top\). The advantage is that we do not need to worry about the typing in variants where \(\pi\) has \(\bot\)s. That also means that we should not use (or trust) the typing result at variants where \(\pi\) has \(\bot\)s. We formally define this relation in Figure 9. For example, since \(\Gamma \vdash 1 : \text{Int}\) we have \(A(\top, \bot) ; \Gamma \vdash A(1, \text{True}) : \text{Int}\), even though \(\text{True}\) does not have the type \(\text{Int}\). We can also generalize this idea to other operations, such as \(\text{dom}\) and \(\text{cod}\), again defined in Figure 9.

As shown in the rule PATUNARY, we can also use patterns to constrain unary functions so that they need to be defined for where only the pattern have \(\top\). In the rule, \(\text{op}\) could be instantiated to any unary functions, such as \(\text{dom}\) and \(\text{cod}\). We use the following function
Migrating Gradual Types

\[ \text{dom} (M_1 \rightarrow M_2) = M_1 \quad \text{dom} (_) = \ast \quad \text{dom} (d | M_1, M_2) = d \text{dom} (M_1), \text{dom} (M_2) \]

The function \( \text{dom} \) is defined for three cases and is undefined for all other inputs.

For example \( \text{dom} (\text{Int} \rightarrow \text{Bool}) = \text{Int} \) but \( \text{dom} (\text{Int}) \) is undefined. How about \( \text{dom} (A(\text{Int} \rightarrow \text{Bool}, \text{Int})) \)? We can observe that it is defined for the first alternative but not the second alternative. In such case, we can constrain \( \text{dom} \) with a pattern to indicate that the function does not need to be defined for all alternatives of variations.

For our example, we can use the pattern \( A(\top, \bot) \) to convey that we only need the first alternative of \( A \) to be defined (because the pattern there is a \( \top \)) while ignore whether the second alternative is defined or not (because the pattern there is a \( \bot \)). With this idea, \( \text{dom}_{A(\top, \bot)} (A(\text{Int} \rightarrow \text{Bool}, \text{Int})) \) is defined in both alternatives of \( A \). Moreover, for the second alternative, we can say the result \( \text{dom} \) is any type because \( \bot \) in that alternative indicates that the typing result will be discarded. Only typing results in variants where typing pattern has \( \top \) are valid and considered.

Similarly, we can define \( \text{cod} \) if we have a function \( \text{cod} \), which we define in Figure 10. The rule \( \text{PatBinary} \) allows us to constrain binary operations or functions in the same way.

Based on the idea of pattern-constrained judgments, we can define the following rule for typing function applications (where \( \text{dom} \) is defined above and \( \text{cod} \) will be defined in Figure 10):

\[
\begin{align*}
\pi; \Gamma \vdash e_1 : M_1 & & \pi; \Gamma \vdash e_2 : M_2 & & \text{dom}_{\pi}(M_1) \approx_{\pi} M_2 \\
\pi; \Gamma \vdash e_1 \ e_2 & : \text{cod}_{\pi}(M_1)
\end{align*}
\]

With this new rule, which accounts for migrational types with type errors, we can revisit the problem of typing widthFunc fixed. Let \( \pi = A(\top, B(\top, \bot)) \). Since widthFunc \( \rightarrow A(\ast, \text{Int} \rightarrow \text{Int}) \) belongs to \( \Gamma \), we have \( \pi; \Gamma \vdash \text{widthFunc} : M \) where \( M = A(\ast, \text{Int} \rightarrow \text{Int}) \). Similarly, we have \( \pi, \Gamma \vdash \text{fixed} : B(\ast, \text{Bool}) \). Next, \( \text{dom}_{\pi}(M) = A(\ast, B) \).

As we have seen earlier, \( A(\ast, \text{Int}) \approx_{\pi} B(\ast, \text{Bool}) \). Thus, all the premises of the application rule are satisfied, and we can derive \( \pi; \Gamma \vdash \text{widthFunc \ fixed} : A(\ast, \text{Int}) \). Based on the result pattern, we should not trust the typing information at the variant \( \{A.2, B.2\} \) since \( [\pi]_{\{A.2, B.2\}} \approx_{\pi} \bot \).

While pattern-constrained judgments simplify the presentation, we still face the challenge of finding 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 \( A(\top, B(\top, \bot)) \) 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 10. 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 \( \Gamma \) constrained by the pattern \( \pi \). The mapping \( \Omega \) collects the types that will be assigned to parameters.
John Peter Campora III, Sheng Chen, Martin Erwig, and Eric Walkingshaw

\[ \pi; \Gamma \vdash e : M \mid \Omega \]

\[ \text{CON} \quad c \text{ is of type } \gamma \]
\[ \pi; \Gamma \vdash c : \gamma \mid \emptyset \]

\[ \text{VAR} \quad x \rightarrow M \in \Gamma \]
\[ \pi; \Gamma \vdash x : M \mid \emptyset \]

\[ \text{ABS} \quad \pi; \Gamma, x \rightarrow V \vdash e : M \mid \Omega \]
\[ \pi; \Gamma \vdash \lambda x.e : V \rightarrow M \mid \Omega \]

\[ \text{ABSDYN} \quad \pi; \Gamma, x \rightarrow d(\star, V) \vdash e : M \mid \Omega \quad d \text{ fresh} \]
\[ \pi; \Gamma \vdash \lambda x : \star.e : d(\star, V) \rightarrow M \mid \Omega \cup \{x \rightarrow V\} \]

\[ \text{APP} \quad \pi; \Gamma \vdash e_1 : M_1 \mid \Omega_1 \quad \pi; \Gamma \vdash e_2 : M_2 \mid \Omega_2 \]
\[ dom(M_1) \approx_{\pi} M_2 \quad M_3 = cod_{\pi}(M_1) \]
\[ \pi; \Gamma \vdash e_1 e_2 : M_3 \mid \Omega_1 \cup \Omega_2 \]

\[ \text{WEAKEN} \quad \pi; \Gamma \vdash e : M \mid \Omega \]
\[ \pi_1 \leq \pi \]
\[ \pi_1 \vdash e : M_1 \mid \Omega \]

\[ \text{dom}(M_1 \rightarrow M_2) = M_1 \quad \text{cod}(M_1 \rightarrow M_2) = M_2 \]
\[ \text{dom}(\star) = \star \quad \text{cod}(\star) = \star \]
\[ \text{dom}(d M_1, M_2) = d \text{ dom}(M_1), \text{dom}(M_2) \quad \text{cod}(d M_1, M_2) = d \text{ cod}(M_1), \text{cod}(M_2) \]
\[ M \cap M = M \quad M_1 \cap_\pi M_2 \cap_\pi M_3 = (M_1 \cap_\pi M_2) \cap_\pi (M_1 \cap_\pi M_3) \]
\[ \star \cap M = M \quad d M_1, M_2 \cap M = d M_1 \cap M, M_2 \cap M \]
\[ M \cap \star = M \quad G \cap d M_1, M_2 = G \cap d M_1, G \cap M_2 \]

\[ \text{PAT-OK} \quad \pi \leq \top \]
\[ \bot \leq \pi \]
\[ \pi_1 \leq \pi_2 \quad \pi_2 \leq \pi_3 \]
\[ \pi_1 \leq \pi \]

\[ \text{PAT-ERR} \quad \pi_1 \leq \pi_3 \]

\[ \text{PAT-TRANS} \quad \pi_1 \leq \pi_2 \quad \pi_2 \leq \pi_3 \]
\[ \pi_1 \leq \pi_3 \]

\[ \text{PAT-SINCH} \quad \pi_1 \leq \pi_2 \quad \pi_1 \leq \pi_3 \]
\[ \pi_1 \leq d(\pi_2, \pi_3) \]

\[ \text{PAT-CHEC} \quad \pi_1 \leq \pi_3 \quad \pi_2 \leq \pi_3 \]
\[ d(\pi_1, \pi_2) \leq \pi_3 \]

\[ \text{PAT-CHECCHC} \quad \pi_1 \leq \pi_3 \quad \pi_2 \leq \pi_4 \]
\[ d(\pi_1, \pi_2) \leq d(\pi_3, \pi_4) \]

Fig. 10: Typing rules. The operations \text{dom}, \text{cod}, and \cap are undefined for cases that are not listed here. The process for obtaining \text{dom}_{\pi} from \text{dom} is detailed in Section 4.3. The operations \text{cod}_{\pi} and \cap_{\pi} can be obtained similarly through Figure 9.

if their \star s are removed. We assume that parameter names from different functions are
uniquely identified in the domain of \Omega. The goal of \Omega is to connect the typing rules here
with those from Figure 4. We discuss this aspect in more detail in Section 4.5 where we
investigate the properties of the type system.

The rules for constants (\text{CON}) and variables (\text{VAR}) are straightforward. They hold
for arbitrary patterns \pi because constants and bound variables are always well typed.
Moreover, since the types remain unchanged, \Omega is always \emptyset. The rule \text{ABS} for an
abstraction whose parameter is not annotated with \star is conventional. In rule \text{ABSDYN} for
an abstraction whose parameter is annotated with \star, we assign the parameter a choice type
where the first alternative is \star implying that the \star is kept and the second alternative can be
any type for the body to be well typed. As a result, when variations are first introduced, their
first alternatives are \star s. This change information is recorded by extending the \Omega returned
from typing the body of the abstraction.
Migrating Gradual Types

The $\text{App}$ 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 operations $\text{dom}_2$ and $\text{cod}_2$ can be obtained from $\text{dom}$ and $\text{cod}$ respectively using the idea of pattern-constrained operations discussed in Section 4.3.

The rule $\text{IF}$ types conditionals; it relies on an extended version of the meet operation ($\sqcap$) from Figure 4 that also handles choices. The definition $\sqcap_\pi$ can be obtained from Figure 9 by instantiating the $\text{op}$ in rule $\text{PAT_BINARY}$ with $\sqcap$. In Section 4.3, we gave a detailed example of deriving $\text{dom}_2$ from $\text{dom}$ and $\sqcap_\pi$ can be derived from $\sqcap$ similarly.

The $\text{WEAKEN}$ 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 $\pi_1 \sqsubseteq \pi_2$ in the premise specifies that its arguments must be the same for places where $\pi_1$ has $\top$ values. A typing pattern $\pi_1$ is less defined than $\pi_2$ if it contains $\bot$ values at least everywhere $\pi_2$ does. The purpose of $\text{WEAKEN}$ is to make the typing process compositional. Without this rule, the whole typing derivation must use the same $\pi$. 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. To illustrate, consider typing an application $e_1 \ e_2$. It is likely that $e_1$ and $e_2$ will contain errors at different variants, and thus the typing patterns for typing them will be different. Without $\text{WEAKEN}$, we should use a single pattern for typing these two subexpressions. With $\text{WEAKEN}$, we can use different patterns for typing subexpressions, and before typing the application itself we can apply $\text{WEAKEN}$ to the typing derivation for either or both $e_1$ and $e_2$ to make their patterns the same. After that, we can apply the $\text{APP}$ rule.

The less-defined relation on patterns, written as $\pi_1 \leq \pi_2$, is formally defined in Figure 10.

The rules $\text{PAT-OK}$ and $\text{PAT-ERR}$ define that any pattern is less defined than $\top$ and more defined than $\bot$. The rule $\text{PAT-TRANS}$ defines that the relation is transitive. The last three rules handle variational patterns. The rule $\text{PAT-SINCHIC}$ states that a pattern is less-defined than a variational pattern if it is less-defined than both alternatives of the variational pattern. The rule $\text{PAT-CHCISING}$ states that a variational pattern is less-defined than a pattern if both alternatives are. Finally, the rule $\text{PAT-CHCCHIC}$ says that two variational patterns satisfy the less-defined relation if their corresponding alternatives do.

4.5 Properties

This subsection investigates the properties of the type system. Since the goal of migrational typing in Figure 10 is to type all possible programs that remove $*$s for a given program at once, we want to investigate whether migrational typing does it currently for individual programs and whether it indeed types all programs that remove $*$s. To this end, we consider the relationship of the rules for migrational typing in Figure 10 and the original rules for gradual typing in Figure 4. We also consider the relation between different typing derivations $\pi, \Gamma \vdash e : M | \Omega$ when different $\pi$s and $M$s are used for the same $\Gamma$ and $e$, which addresses challenge C3 (best typing) from Section 3.

We start by introducing some notation. We say a decision $\delta$ is complete for an expression $e$ if it contains $d.1$ or $d.2$ for each $d$ created while typing $e$. For $\pi$, a decision $\delta$ is complete if $[\pi]_\delta$ yields $\top$ or $\bot$. Note that a complete decision for $\pi$ may not be complete for the expression since patterns compactly represent where typing succeeds and where it
For example, for encoded in migrational typing is the same as by typing the program with ITGL. Where the typing pattern is valid, it assigns the same types to all the programs in the domain $d$ such that $d \in \delta$.

The notions of decisions ($\delta$), variational statifier ($\Omega$), and statifier ($\omega$) are closely related. Specifically, during typing, for each dynamic parameter $x$, $\Omega$ includes a mapping $x \mapsto V$, where $V$ is the type that will be assigned to the parameter once its $\star$ annotation is removed. Therefore, given $\Omega$ and $\delta$, we can generate a statifier as follows, where $\text{cht}(x)$ returns the name of the choice created for $x$.

$$\text{statifierForDesc}(\Omega, \delta) = \{x \mapsto [V]_\delta \mid x \mapsto V \in \Omega \land \text{cht}(x) \in \delta[2]\}$$

For example, let

$$\Omega_d = \{\text{fixed} \mapsto \text{Bool}, \text{widthFunc} \mapsto \text{Int} \to \text{Int}\} \quad \delta_d = \{A.2, B.1\}$$

then $\text{statifierForDesc}(\Omega_d, \delta_d) = \{\text{fixed} \mapsto \text{Bool}\}$.

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

$$\begin{array}{c}
T_1 \sqsubseteq T_2 \quad \star \sqsubseteq \star \quad \star \sqsubseteq G \\
G_1 \sqsubseteq G_3 \quad G_2 \sqsubseteq G_4
\end{array}$$

For example, let $G_1 \sqsubseteq G_2$ be written as $G_1 \preceq G_2$ if $G_1 \sqsubseteq G_2$ or $G_1 = \theta_2(G_2)$ for some $\theta_2$. Intuitively, $G_1 \preceq G_2$ if $G_2$ is equally or more static than $G_1$ or they are equally static and for any static part in $G_1$, $G_2$ has the same static type or a type variable. For example, we have $\star \mapsto \alpha \preceq \text{Int} \to \text{Int}$ and $\text{Int} \to \text{Int} \preceq \text{Int} \to \alpha$.

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** ($\star$ removal soundness)

If $\pi;\Gamma \vdash e : M | \Omega$, then $\forall \delta : [\pi]_\delta = \top \Rightarrow \text{statifierForDesc}(\Omega, \delta) ; [\Gamma]_\delta \vdash_G e : [M]_\delta$.

This theorem states that, for any removal of $\star$ annotations, the typing result encoded in migrational typing is the same as by typing the program with ITGL. For example, for $\pi'_a = A(\top, B(\top, \bot))$ we get $\pi'_a ; \Gamma \vdash \text{width} : M_a | \Omega_a$, where $M_a = A(\star, \text{Bool}) \to B(\star, \text{Int} \to \text{Int}) \to B(\star, \text{Int})$ and $\Omega_a$ is as defined earlier. We can verify $\text{statifierForDesc}(\Omega_a, \delta_a) ; \Gamma \vdash_G \text{width} : \text{Bool} \to \star \to \star$ and $[M_a]_{\delta_a} = \text{Bool} \to \star \to \star$, where $\delta_a$ is as defined earlier.

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

**Theorem 5** ($\star$ removal completeness)

If $\omega ; \Gamma \vdash_G e : G$, then there exists some typing $\pi ; \Gamma \vdash e : M | \Omega$ such that $[\pi]_\delta = \top$, $[M]_\delta = G$, and $\text{statifierForDesc}(\Omega, \delta) = \omega$ for some $\delta$. 

Migrating Gradual Types

We can observe that for a given expression, there may be multiple typing derivations based on the typing rules in Figure 10. The reason is that, for example, the variational types used for typing the same AbsDyn in different typings could be different. Particularly, we want to know if there exists a best typing derivation that is more static and more defined (the corresponding typing pattern contains \( \perp \) in fewest variants) than all other derivations. Fortunately, this is indeed the case (Lemma 2). We next investigate the relation between different typings. In Lemma 1, we will show that different typings can be combined to make the result as correct as possible (that is, to minimize \( \perp \)s in the result pattern). In Lemma 2, we show different typing can be combined to be made 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 \( \Omega \). As a result, the domain of \( \Omega \)s in different typings are the same. However, the ranges could be different because different typings may use different \( V \)s in AbsDyn.

**Lemma 1**

If \( \pi_1; \Gamma \vdash e : M | \Omega_1 \) and \( \pi_2; \Gamma \vdash e : M | \Omega_2 \), then there is some typing \( \pi; \Gamma \vdash e : M | \Omega \) such that \( \pi_1 \leq \pi \) and \( \pi_2 \leq \pi \).

The following lemma states that we can always find a better (in the sense of the better relation defined at the beginning of this section, in Page 24) variational statifier and typing for any expression.

**Lemma 2**

If \( \pi; \Gamma \vdash e : M_1 | \Omega_1 \) and \( \pi; \Gamma \vdash e : M_2 | \Omega_2 \), then there is some typing \( \pi; \Gamma \vdash e : M | \Omega \) such that \( \forall \delta, [\pi]_\delta = \top \Rightarrow [M_1]_\delta \leq [M]_\delta \land [M_2]_\delta \leq [M]_\delta \land \text{statifierForDesc}(\Omega_1, \delta) \leq \text{statifierForDesc}(\Omega, \delta) \land \text{statifierForDesc}(\Omega_2, \delta) \leq \text{statifierForDesc}(\Omega, \delta) \).

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 6 (MGSM Typing)**

For any \( e \) and \( \Gamma \), there is a MGSM typing \( \pi; \Gamma \vdash e : M | \Omega \) such that for any \( \pi_1; \Gamma \vdash e : M_1 | \Omega_1, \forall \delta, [\pi_1]_\delta = \top \Rightarrow [\pi]_\delta = \top \land [M_1]_\delta \leq [M]_\delta \).

**Proof of Theorem 6**

The proof of the best typing is a direct consequence of Lemma 1 and Lemma 2, meaning that we can produce a most precise and general typing and then give a most defined pattern to it. \( \square \)

To illustrate the use of Theorem 6, the MGSM typing for width is \( \pi_0; \Gamma \vdash \text{width} : M_0 | \Omega_0 \), where

\[
\Omega_0 = \{ \text{fixed} \rightarrow \text{Bool}, \text{widthFunc} \rightarrow \text{Int} \rightarrow \beta \} \\
M_0 = A(\top, B(\top, \perp)) \\
\pi_0 = A(\top, B(\top, \perp))
\]

Theorem 6 implies that while an infinite number of typings may be derived (due to the \( \perp \) 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 (migration extraction) 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 an MGSM typing in Section 5.2.

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

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

We say an eliminator $\delta$ is valid if $|\pi|_\delta = \top$ where $\pi$ should be clear from the context.

We will use $\delta^\ast$ to denote valid eliminators. For example, let

$$\delta_1^\ast = \{A.1,B.1\} \quad \delta_2^\ast = \{A.1,B.2\} \quad \delta_3^\ast = \{A.2,B.1\} \quad \delta_4 = \{A.2,B.2\}$$

then $\delta_1^\ast \gg \delta_2^\ast$ and $\delta_3^\ast \gg \delta_4$, but $\delta_2^\ast \not\gg \delta_1^\ast$. The eliminators $\delta_1^\ast$, $\delta_2^\ast$, and $\delta_3^\ast$ are valid, while $\delta_4$ is not, with respect to $\pi_0$ 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 computing 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 do not have to consider even all of these). For example, the pattern $\pi_0$ from Section 4.5 for rowAtI has only 5 eliminators while the expression itself has 32. As another example, from the pattern $\pi_0$, defined at the end of Section 4.5 (page 25), we can see that $\delta_{\text{width}}^\ast = \{A.1\}$ compactly represents $\delta_1^\ast$ and $\delta_2^\ast$ 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 $\delta_4$ is already the strictest for $\pi_0$. 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 $\pi_0$, 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 answers this question.

Theorem 7 (Error Irrecoverability)

Let $\pi,\Gamma \vdash e : M | \Omega$ be an MGSM typing for $e$ and $\Gamma$. If $|\pi|_\delta = \bot$, then $\forall \delta_1. \delta \gg \delta_1 \Rightarrow |\pi|_{\delta_1} = \bot$.

This theorem implies that we can simply ignore invalid eliminators, and focus on valid ones, since all invalid eliminators lead to ill typed expressions.
Migrating Gradual Types

Proof

Proof by contradiction. Assume there is some $\delta_1$ such that $\delta \gg \delta_1$ but $|\pi|_{\delta_1} = \top$. According to Theorem 4, we have $\text{statifierForDesc}(\Omega, \delta_1); [\Gamma]_{\delta} \vdash_{\text{GC}} e : [M]_{\delta_1}$, which means that $e$ is well typed under the statifier $\text{statifierForDesc}(\Omega, \delta_1)$. Based on the definition of statifier generation (Section 4.5), we know that $\delta \gg \delta_1$ implies that $\text{statifierForDesc}(\Omega, \delta) \subseteq \text{statifierForDesc}(\Omega, \delta_1)$. Therefore, applying $\text{statifierForDesc}(\Omega, \delta)$ to $e$ yields a less static expression than $\text{statifierForDesc}(\Omega, \delta_1)$ does. Based on the static gradual guarantee for ITGL (Miyazaki et al., 2019), the typing relation $\text{statifierForDesc}(\Omega, \delta); [\Gamma]_{\delta} \vdash_{\text{GC}} e : [M]_{\delta}$ is satisfied. According to Theorem 6, this implies that $|\pi|_{\delta} = \top$, which contradicts our condition that $|\pi|_{\delta} = \bot$. Therefore, there is no $\delta_1$ such that $\delta \gg \delta_1$ but $|\pi|_{\delta_1} = \top$ exists, completing the proof.

A valid eliminator for the typing pattern corresponds to potentially many valid eliminators for the expression. We say that a valid pattern eliminator $\delta_1$ covers a valid expression eliminator $\delta_2$ if $\delta_1 \subseteq \delta_2$. Among all the expression eliminators covered by a pattern eliminator, one is the strictest. For example, the eliminator $\delta_{\text{in}}^v$ for pattern $\pi_0$ covers the eliminators $\delta_{\text{in}}^y$ and $\delta_{\text{in}}^{\text{match}}$ for typing width, and $\delta_{\text{in}}^y$ is the strictest. As another example, the valid eliminator $\delta_{\text{in}}^{\text{match}} = \{A.1,E.1\}$ for pattern $\pi_a$ covers eight valid eliminators (two options for each of the three choice names that do not appear in the pattern) for typing $\text{rowAtI}$, 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. The following theorem 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 $\pi_b$, neither of the valid eliminators $\delta_{\text{in}}^v$ or $\delta_{\text{in}}^y$ is stricter than the other. Similarly, for pattern $\pi_c$, 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 9 (Eliminator Incomparability)
We consider several cases.

- \( \delta_1^v = \{d_1.1,d_2.1\} \) and \( \delta_2^v = \{d_1.1,d_2.2\} \) or \( \{d_1.2,d_2.1\} \) or \( \{d_1.2,d_2.2\} \). Based on \( \delta_2^v \), \( \delta_3^v = \{d_1.1,d_2.2\} \) is a valid eliminator based on the inverse of the implication in Theorem 7. From \( \delta_2^v \) and \( \delta_3^v \), we can infer that both alternatives of \( d_2 \) are \( \top \), meaning that it is an idempotent variation and \( \pi \) is not in normal form.
- \( \delta_1^v = \{d_1.1,d_2.2\} \), \( \{d_1.2,d_2.1\} \), or \( \{d_1.2,d_2.2\} \). The reasoning is similar to the previous case by showing that the variation \( d_2 \) is idempotent.
- \( \delta_1^v = \{d_1.1\} \) and \( \delta_2^v = \{d_1.2,d_2.1\} \). The decision \( \delta = \{d_1.2,d_2.2\} \) satisfies \( \delta^v \gg \delta \wedge \delta_2^v \gg \delta \). If \( \delta \) is a valid eliminator, then we can again show that \( d_2 \) is idempotent, a contradiction that \( \pi \) is in normal form.

We could swap the assignments to \( \delta_1^v \) and \( \delta_2^v \), but this will yield the same proof result.

It follows from the theorem that for any two valid eliminators \( \delta_1^v \) and \( \delta_2^v \) for \( \pi_1 \), \( \delta_1^v \gg \delta_2^v \) and \( \delta_2^v \gg \delta_1^v \). Two eliminators that are incomparable with respect to \( \gg \) will remove *s for different parameters for the same expression, leading to types that are incomparable by \( \preceq \) (defined in Section 4), and thus incomparable by \( \preceq \). For example, since \( \delta_2^v \gg \delta_3^v \) and \( \delta_1^v \gg \delta_3^v \), we have \( G_b \nsubseteq G_c \) and \( G_c \nsubseteq G_b \), where \( G_b = [M_b]_{\delta_2^v} = * \rightarrow (\text{Int} \rightarrow \beta) \rightarrow \beta \) and \( G_c = [M_b]_{\delta_3^v} = \text{Bool} \rightarrow * \rightarrow * \).

Combining Theorems 8 and 9, yields the following result about finding most static migrations. We develop an algorithm for extracting such migrations in Section 5.2.

**Theorem 10 (Uniqueness of most static migrations)** Let \( \pi, \Gamma \vdash e : M \mid \Omega \) be the MGSM typing for \( e \) and \( \Gamma \), and \( \pi \) is in normal form. Then the number of most static migrations for \( e \) equals the number of valid eliminators for \( \pi \).

**Proof of Theorem 10**
The proof follows directly from Theorem 9 and Theorem 8. Theorem 9 implies that complete decisions are not comparable and no other complete decisions are better than them. Theorem 8 implies that tighter selectors yields more precise types. By definition, each complete decision yields a most static migration, since no types better than those produced by complete decisions can be assigned to the expression.

It follows from the theorem that \( e \) has a unique most static migration if \( \pi_1 \) has only one valid eliminator.

### 5.2 Extracting Most Static Migrations

The most static migrations for a program are identified by valid eliminators that describe whether to pick the * annotation or the inferred type for each parameter. We compute this
Migrating Gradual Types

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 10 to find most static migrations. We use the following rules to simplify patterns to normal forms.

\[ d(\pi, \pi) \leadsto \pi \quad d(\pi_1, \pi_2) \leadsto d([\pi_1]_{d.1}, [\pi_2]_{d.2}) \quad \pi_1 \leadsto \pi_2 \]

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 \( \pi_c = A(E(B(\top, \bot), B(\bot, \bot))) \) to pattern \( \pi_s \) from Section 4.5.

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

\( ve(\top) = \{\emptyset\} \quad ve(\bot) = \emptyset \quad ve(d(\pi_1, \pi_2)) = \{\{d.1\} \cup l \mid l \in ve(\pi_1)\} \cup \{\{d.2\} \cup r \mid r \in ve(\pi_2)\} \)

To illustrate the definition of \( ve \), we consider the calculation process for the pattern \( A(\top, \bot) \).

\( ve(A(\top, \bot)) = \{\{A.1\} \cup l \mid l \in ve(\top)\} \cup \{\{A.2\} \cup r \mid r \in ve(\bot)\} = \{\{A.1\} \cup \emptyset \} = \{\{A.1\}\} \). This means that the set of valid eliminators for \( A(\top, \bot) \) contains only one element: \( \{A.1\} \).

Similarly, \( ve(A(\bot, \top)) = \{\{A.2\}\} \). As another example, \( ve(\pi_s) \) yields \( \{\delta^s_1, \delta^s_2\} \), where \( \delta^s_1 = \{A.1,E.1\} \) and \( \delta^s_2 = \{A.2,B.1,E.1\} \).

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

\[ expand(\delta, \bar{D}) = \delta \cup \{d.2 \mid d \in \bar{D} \land d.1 \notin \delta\} \]

For example, the set of choice names \( \bar{D} \) for typing \( rowAtI \) is \( \{A,B,D,E,F\} \),

and \( expand(\delta^s_1, \bar{D}) \) yields \( \{A.1,E.1,B.2,D.2,F.2\} \) and \( expand(\delta^s_2, \bar{D}) \) 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 \( \ast \) annotation for \( widthFunc \), \( table \), and \( i \), while the other best eliminator removes the \( \ast \) annotation for \( fixed \), \( table \), and \( i \).

Formally, given an expression \( e \) and its MGSM typing \( \pi; \Gamma \vdash e : M | \Omega \), then for any expanded valid eliminator \( \delta^s \), we can generate the most static migration using \( statifierForDesc(\Omega, \delta^s) \), defined in Page 24.

Overall, these three steps provide a simple way to extract the most static migration from an MGSM typing. In Section 10, we show that these steps lead to an efficient implementation. Usually, the normal form of a typing pattern is small and has only a few valid eliminators. For example, if the program is still well typed after removing all \( \ast \) 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 \( \ast \) annotation is removed, then there is again just one valid eliminator.
John Peter Campora III, Sheng Chen, Martin Erwig, and Eric Walkingshaw

ConC $\Gamma \vdash C : y | e$

VarC $x : M \in \Gamma$

Abs $\Gamma, x \mapsto \alpha \vdash C : e : M | C \quad \alpha$ fresh

AbsDyn $\Gamma, x \mapsto d(\ast, \alpha) \vdash C : e : M | C \quad d$ fresh

AppC codCst $(M_1) \mapsto (M_1, C_1)\quad$ domCst $(M_1, M_2) \mapsto C_4\quad C = C_1 \land C_2 \land C_3 \land C_4$

IfC $\Gamma \vdash C : e_1 : M_1 | C_1 \quad \Gamma \vdash C : e_2 : M_2 | C_2$

$M_2 \land M_3 \mapsto (M_4, C_4)\quad C = C_1 \land C_2 \land C_3 \land C_4 \land M_1 \approx \mathbb{B}$

$\Gamma \vdash C$ if $e_1$ then $e_2$ else $e_3 : M_4 | C$

Fig. 11: Constraint generation rules

$\text{domCst}(\ast, M) \mapsto \varepsilon$

$\text{domCst}(\alpha, M) \mapsto \alpha \approx M \mapsto \kappa_2$

$\text{domCst}(M_1 \rightarrow M_2, M) \mapsto M_1 \approx M \rightarrow d(\text{domCst}(M_1, M), \text{domCst}(M_2, M))$

$\text{domCst}(\_, \_) \mapsto \text{Fail}$

$\text{codCst}(\ast) \mapsto (\ast, \varepsilon)$

$\text{codCst}(\alpha) \mapsto (\kappa_2, \alpha \approx \kappa_1 \mapsto \kappa_2)$

$\text{codCst}(M_1 \rightarrow M_2) \mapsto (M_2, \varepsilon)$

$\text{codCst}(d(M_1, M_2)) \mapsto d(\text{codCst}(M_1), \text{codCst}(M_2))$

$\alpha \cap M \mapsto (\alpha, \alpha \approx \kappa_1)$

$d(M_1, M_2) \cap M \mapsto d(M_1 \cap M, M_2 \cap M)$

$M_1 \cap \alpha \mapsto (\alpha, \alpha \approx \kappa_1)$

$d(M_1 \cap M, M_2 \cap M)$

$\ast \cap M \mapsto (M, \varepsilon)$

$d(M_1 \cap M_2) \cap M \mapsto d(M_1 \cap M_2, C_1 \land C_2)$

$M \cap \ast \mapsto (M, \varepsilon)$

$M_1 \cap M_2 \cap M_3 \mapsto (M_1 \cap M_2, C_1 \land C_2)$

$\bot \cap \ast \mapsto (\kappa, \text{Fail})$ where $M_1 \cap M_2 \cap M_3 \mapsto (M_1, C_1)$

$M_2 \cap M_3 \mapsto (M_2, C_2)$

Fig. 12: Auxiliary constraint generation functions.

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

6 Constraint Generation

The constraint generation rules are presented in Figure 11. The judgment $\Gamma \vdash C e : M | C$ states that under $\Gamma$, the expression $e$ has type $M$ when the constraint $C$ is solved. Accordingly, $e$ and $\Gamma$ are inputs, while $M$ and $C$ are outputs. Note that we now omit the statifier $\Omega$ in constraint judgments since it is not needed for type inference. We also omit $\pi$ since $\pi$ is an input in the declarative typing but will be computed through solving constraints generated here. Constraint solving will be discussed in Section 7. The syntax of constraints are as follows:

$$C ::= M_1 \approx M_2 | C \land C | d(C, C) | \varepsilon | \text{Fail}$$
Migrating Gradual Types

The first form represents type compatibility constraints. Often it is the case that two types are only partially compatible. Note, when $M_1 \approx^1 M_2$ is solved, it is not necessary that $M_1$ and $M_2$ are compatible everywhere. As a result, constraint solving result includes a typing pattern, which indicates where $M_1$ and $M_2$ are indeed compatible. The constraint $C_1 \land C_2$ defines the conjunction of two constraints $C_1$ and $C_2$, while the constraint $d(C_1, C_2)$ defines a choice between two constraints. The constraint $\varepsilon$ represents an empty constraint. This is needed to represent a judgment where no constraints are generated.

Finally, the constraint $\text{Fail}$ represents a constraint that, when solved, always leads to a failure. Such a constraint is needed when, for example, $\text{dom} (\text{Int})$ is calculated during the constraint generation process. As Int is not a function type, $\text{dom} (\text{Int})$ will always fail. We generate a $\text{Fail}$ to communicate this failure to the constraint solver. The constraint $\text{Fail}$ was absent from the original paper (Campora et al., 2018a). Without it, that work outputs a typing pattern and returns a $\bot$ as the typing pattern to denote that certain constraint will definitely fail to solve.

A drawback of that approach is that both constraint generation and constraint solving output typing patterns, and these patterns have to be combined into a single pattern, which is one part of type inference result. That work used the notion of “pattern placeholders”, which are introduced during constraint generation and will be plugged in with concrete patterns during constraint solving. The introduction of $\text{Fail}$ simplifies the handling of patterns. Specifically, only constraint solving outputs a pattern, and we do not need the notion of “pattern placeholders”. Also, the typing pattern has no longer to be part of the constraint generation judgment. Moreover, with $\text{Fail}$ we have simplified the judgments and definitions of several auxiliary functions (Figures 11 and 12) in this version.

We now walk through each constraint generation rule. The rule $\text{ConC}$, generating constraints for constants, has a very similar form to $\text{Con}$ in Figure 10. The rules $\text{VarC}$ for variable references is similar to $\text{Var}$ and, like $\text{ConC}$, generates the empty constraint.

The rule $\text{AnsDynC}$ generates constraints for abstractions with dynamic parameters. It helps facilitate migration by creating a fresh choice type with a left alternative containing $\ast$ 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 rules $\text{AppC}$ and $\text{IfC}$ are more involved because constraints from premises have to be combined. The rules $\text{AppC}$ and $\text{IfC}$ use many auxiliary functions to generate constraints. The functions, defined in Figure 12, take the form: $\text{domCst}(M_1, M_2) \leftrightarrow C$, $\text{codCst}(M_1) \leftrightarrow (M_2, C)$, and $M_1 \sqcap M_2 \leftrightarrow (M_1, C)$, where the objects to the left of $\leftrightarrow$ are inputs and those to the right are outputs. Essentially, they implement the $\text{dom}$, $\text{cod}$, and $\sqcap$ operations defined for the declarative type system in Figure 10. Note, in these functions $\kappa$ denote fresh type variables. We will use such variables in this and next sections.

We illustrate $\text{domCst}$ by considering the example $\text{domCst}(A(\ast, \alpha, \text{Int}))$. Since the first argument is a choice type, $\text{domCst}$ proceeds recursively call on each alternative of $A$, leading to two subproblems $\text{domCst}(\ast, \text{Int})$ and $\text{domCst}(\alpha, \text{Int})$. The first subproblem is handled by the case for $\ast$, which immediately returns $\varepsilon$, meaning that no further constraints need to be solved. The second subproblem is handled by the case of $\text{domCst}$ for type variables. Since $\text{dom}$ always expects a function type, the constraint $\alpha \approx^1 \text{Int} \rightarrow \kappa_2$ is generated. The constraints for subproblems are combined together with the choice $A$, yielding the final constraint $A(\varepsilon, \alpha \approx^1 \text{Int} \rightarrow \kappa_2)$. 
The following soundness (Theorem 11) and completeness (Theorem 12) theorems state that the constraint generation rules correspond to the declarative typing rules presented in Figure 10. In particular, Theorem 12 implies that constraint generation finds the MGSM used the term guess-free). \((\theta, \pi)\) is sound for a constraint of the form \(M_1 \approx \theta (M_2)\) if \(\theta (M_1) \approx \pi \theta (M_2)\), is sound for a constraint \(\pi \wedge C_1 \wedge C_2\) or \(d (C_1, C_2)\) if it is sound for both \(C_1\) and \(C_2\), is sound for \(\text{Fail}\) if \(\pi\) is \(\bot\), and is always sound for \(\epsilon\). In Section 7, we provide a unification algorithm that generates solutions with these desired properties.

**Theorem 11 (Soundness of Constraint Generation)**

If \(\Gamma \vdash e : M \mid C\), then \(\pi; \theta (\Gamma) \vdash e : \theta (M) \mid \Omega\) for some \(\Omega\), where \((\theta, \pi)\) is a sound solution for \(C\).

**Theorem 12 (Completeness of Constraint Generation)**

If \(\pi; \theta (\Gamma) \vdash e : M \mid \Omega\) then \(\Gamma \vdash e : M_1 \mid C\) such that \(\pi \leq \pi_1\), \(\forall \delta. [\pi_1]_\delta = \top \Rightarrow [\pi_1]_\delta = \top \land \mid M_1 \mid_\delta \leq [\theta (M_1)]_\delta \land [\pi]_\delta = [\theta']_\delta \circ [\theta_1]_\delta\) for some \(\theta'\), where \((\theta_1, \pi_1)\) is a sound and most-general solution for \(C\).

In the theorem, we define \([\theta]_\delta = \{\alpha \mapsto [V]_\delta \mid \alpha \mapsto V \in \theta\}\).

**Two constraint generation examples** The following table lists the constraint generation process for the expression \(\lambda x : \ast \cdot \text{succ} (x \text{ True})\). In each row, we list the subexpression visited, the type of that subexpression, and the constraint generated. Assume the fresh choice and variable generated for the parameter are \(A\) and \(\alpha\), respectively.

<table>
<thead>
<tr>
<th>Subexpression</th>
<th>(M) (Type)</th>
<th>(C) (Constraint)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(x)</td>
<td>(A (\ast, \alpha))</td>
<td>(\epsilon)</td>
</tr>
<tr>
<td><code>True</code></td>
<td><code>Bool</code></td>
<td>(\epsilon)</td>
</tr>
<tr>
<td>(x \text{ True})</td>
<td>(A (\ast, \kappa))</td>
<td>(A (\epsilon, C_1 \wedge C_2))</td>
</tr>
<tr>
<td><code>succ</code></td>
<td><code>Int \rightarrow Int</code></td>
<td>(\epsilon)</td>
</tr>
<tr>
<td><code>succ (x True)</code></td>
<td><code>Int</code></td>
<td>(A (\epsilon, C_1 \wedge C_2 \wedge C_4))</td>
</tr>
<tr>
<td>(\lambda x : \ast \cdot \text{succ} (x \text{ True}))</td>
<td>(A (\ast, \alpha) \rightarrow \text{Int})</td>
<td>(A (\epsilon, C_1 \wedge C_2 \wedge C_4))</td>
</tr>
</tbody>
</table>

\(C_1 = \alpha \approx \gamma \kappa_1 \rightarrow \kappa_2\) \(C_2 = \alpha \approx \gamma \text{ Bool} \rightarrow \kappa_4\) \(C_4 = \text{Int} \approx \gamma \kappa_2\)

The constraints \(C_1\) and \(C_2\) are generated from the third and fourth premises of APPC for typing `x True`, respectively. The constraint \(C_4\) is generated from the fourth premise of APPC for handling the application `succ (x True)`.

Continuing from the fifth row of the table above, the following table lists additional constraints that will be generated from the expression \(\lambda x : \ast \cdot x (\text{succ} (x \text{ True}))\).

<table>
<thead>
<tr>
<th>Subexpression</th>
<th>(M) (Type)</th>
<th>(C) (Constraint)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(x)</td>
<td>(A (\ast, \alpha))</td>
<td>(\epsilon)</td>
</tr>
<tr>
<td><code>x (\text{succ} (x \text{ True}))</code></td>
<td>`A (\ast, \kappa))</td>
<td>`A (\epsilon, C_1 \wedge C_2 \wedge C_4 \wedge C_5 \wedge C_6))</td>
</tr>
<tr>
<td>(\lambda x : \ast \cdot x (\text{succ} (x \text{ True})))</td>
<td>(A (\ast, \alpha) \rightarrow A (\ast, \kappa))</td>
<td>`A (\epsilon, C_1 \wedge C_2 \wedge C_4 \wedge C_5 \wedge C_6))</td>
</tr>
</tbody>
</table>

\(C_5 = \alpha \approx \gamma \kappa_5 \rightarrow \kappa_6\) \(C_6 = \alpha \approx \gamma \text{ Int} \rightarrow \kappa_8\)
7 Unification

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

7.1 Solving Compatibility Constraints

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

(i) $\alpha \approx \star \rightarrow \text{Int}$

(ii) $A(\star, \text{Bool}) \approx \text{Int}$

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

(I) $\star$ is compatible with any type (Section 2.1).

(II) Type variables are only substituted by static types (Section 4).

(III) The typing pattern produced must be as defined as possible (Section 4).

Problem (i) helps illustrate rule (II). Intuitively, $\alpha$ should be substituted by a function type whose codomain is $\text{Int}$, 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 the substitutions $\{\kappa_2 \mapsto \text{Int}\} \circ \{\alpha \mapsto \kappa_1 \rightarrow \kappa_2\}$. Since $\kappa_1$ is a fresh type variable that is not mapped to anything, it is unconstrained. In contrast, $\kappa_2$ is mapped to $\text{Int}$. 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, $\star \approx \text{Int}$ and $\text{Bool} \approx \text{Int}$, where $\pi = A(\pi_1, \pi_2)$. According to rule (I), the first constraint is solved successfully and $\pi_1$ is updated to $\top$. The second constraint, however, fails to solve, since $\text{Bool}$ cannot be made compatible with $\text{Int}$, so we update $\pi_2$ to $\bot$. Consequently, we update $\pi$ to $A(\top, \bot)$ to reflect that constraint solving fails in $A$. Choosing instead $\bot$ for $\pi$ would yield a consistent result but would violate rule (III).

7.2 A Unification Algorithm

Figure 13 presents a unification algorithm $U$, which takes a constraint and produces a substitution $\theta$ and a pattern $\pi$. The algorithm can be understood as extending Robinson’s unification algorithm (Robinson, 1965a) to handle variational types and dynamic types and to support error tolerance. To support error tolerance, the unification not only returns a substitution but also a typing pattern. The unification is successful at variants where the pattern has $\top$ and is failed at variants where the pattern has $\bot$. In the algorithm, cases (a) and (a*) deal with dynamic types, cases (c), (d), and (d*) deal with variations. Cases (g) through (j) deal with non-compatibility constraints. Other cases of the algorithm resemble their counterparts in Robinson’s algorithm but still need to account for occurrences of $\star$ and variations.

In the figure, we use the following conventions and helper functions. We use $\kappa$s to denote fresh type variables. The function $\text{choices}(M)$ returns the set of choice names in $M$; $\text{vars}(M)$ returns the set of type variables in $V$. The predicate $\text{hasDyn}(M)$ determines
Fig. 13: A unification algorithm.

whether \( \ast \) occurs anywhere in \( M \). The function \( \text{merge} \) combines the substitutions from solving the subproblems of a choice constraint. For example, given \( d, \theta_1 = \{ \alpha \mapsto \text{Int} \}, \) and \( \theta_2 = \{ \alpha \mapsto \text{Bool} \} \), we have \( \text{merge}(d, \theta_1, \theta_2)(\alpha) = \{ \alpha \mapsto d(\text{Int,Bool}) \} \). Formally, the definition of \( \text{merge} \) (for each \( \alpha \) in \( \theta_1 \cup \theta_2 \)) is:

\[
\text{merge}(d, \theta_1, \theta_2)(\alpha) = d(\text{get}(\alpha, \theta_1), \text{get}(\alpha, \theta_2)) \quad \text{where} \quad \alpha \in \text{dom}(\theta_1) \cup \text{dom}(\theta_2)
\]

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

Intuitively, if \( \alpha \in \text{dom}(\theta) \), then \( \text{get}(\alpha, \theta) \) returns the image of \( \alpha \) in \( \theta \). Otherwise, \( \text{get}(\alpha, \theta) \) returns a fresh type variable. Recall that \( \kappa \) denotes a fresh type variable.

We now briefly walk through each case of \( \mathcal{U} \). Some cases of \( \mathcal{U} \) have dual cases, and names of such cases differ by a \( \ast \). Essentially, the starred version delegates the real solving task to the case without a \( \ast \). Case (a) handles the trivial constraints involving \( \ast \). Such constraints are simply discarded without generating any mapping. We return \( \top \) as the pattern, since \( \ast \) is compatible with any type. More importantly for \( \alpha \approx \ast \), case (a) takes priority over (b), ensuring that the substitution \( \{ \alpha \mapsto \ast \} \) is not generated.

Case (b) unifies a type variable \( \alpha \) with a migrational type \( M \). This case includes many subcases. First, if \( M \) does not contain \( \ast \) and \( \alpha \) does not occur in \( M \), then \( \alpha \) is directly mapped to \( M \). For example, given \( \alpha \approx \ast \{ \text{Int,Bool} \} \), the substitution \( \{ \alpha \mapsto \text{Int,Bool} \} \) is returned, and \( \pi \) is updated to \( \top \). Second, if \( M \) contains variation, the result is computed
Migrating Gradual Types

via case (d). For example, the problem \( \alpha \approx 3 A(\cdot, \Int) \) is transformed into \( A(\alpha, \alpha) \approx 2 \).

Next, if \( M \) is a function type that contains \( \cdot \) and \( \alpha \) does not occur in \( M \), then we transform \( \alpha \) 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. This case essentially solves two constraints, and we will have two typing patterns (\( \pi_1 \) and \( \pi_2 \) in the algorithm). We need to combine them into one. The resulting pattern must be restricted enough to create a valid solving result but well defined enough to give useful information about where constraint solving succeeds.

The operation \( \sqcap \), reproduced from above Lemma 17 for readability, can be viewed as a meet operation over the less defined partial order on typing patterns in Figure 10. It creates the greatest lower bound of two patterns, ensuring that the most defined pattern is used for solving the constraint.

\[
\top \sqcap \pi = \pi \\
\bot \sqcap \pi = \bot \\
d(\pi_1, \pi_2) \sqcap d(\pi_3, \pi_4) = d(\pi_1 \sqcap \pi_3, \pi_2 \sqcap \pi_4)
\]

Back to case (b), if all previous subcases 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 \( A(\cdot, \alpha) \approx 3 A(\Int, \Bool) \). This constraint will be decomposed into \( \ast \approx 3 \Int \) and \( \alpha \approx 3 \Bool \), 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 employs a similar implementation idea as case (c) does. For example, for \( A(\ast, \Int) \approx 3 \Int \), the two smaller constraints to be solved are \( \ast \approx 3 \Int \) and \( \Int \approx 3 \Int \). Case (e) unifies two static types and is delegated to Robinson’s unification algorithm (Robinson, 1965b). Case (f) unifies two function types by unifying their respective argument and return types. Cases (g), (h), (i), and (j) 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), when creating the choice pattern \( d(\pi_1, \pi_2) \), we check if \( \pi_1 \) and \( \pi_2 \) are the same; if so, the choice pattern is replaced by \( \pi_1 \). In the last two cases of \( \sqcap \) in Section 6, we perform the same optimization. After this, the algorithm maintains patterns in normal forms, since the generated constraints do not contain dead alternatives and since the case (d) of \( \forall \) prevents dead alternatives from being introduced.

**Unification examples** In Section 6 we generated two constraints for the expressions \( \lambda x : \ast \succ (x \True) \) and \( \lambda x : \ast x (\succ (x \True)) \). We use these two constraints to illustrate the unification process.

The first constraint is \( A(\varepsilon, C_1 \land C_2 \land C_4) \). For this constraint, case (h) applies, which breaks the variational constraint into two smaller constraints in each alternative and then combine the results from alternatives. The left alternative has the constraint \( \varepsilon \), which will be solved by case (g) with the solution \( (\theta_1, T) \), where \( \theta_1 = \emptyset \). The right alternative has the constraint \( C_1 \land C_2 \land C_4 \). We will repeatedly use case (i) to handle each subconstraint \( C_1 \) through \( C_4 \). Since there are no \( \ast \ast \)s and variations in these constraints, they degenerate to conventional type equality constraints. We can use Robinson’s unification algorithm to...
solve them. The unifier is
\[ \theta_r = \{ \alpha \mapsto \text{Bool} \rightarrow \text{Int}, \kappa_1 \mapsto \text{Bool}, \kappa_2 \mapsto \text{Int}, \kappa_4 \mapsto \text{Int} \} \]

The typing pattern for solving them is \( \top \) as the solving for each constraint returns \( \top \).

After we have the solutions for both alternatives, we will now combine them together. First, the combined typing pattern is \( A(\top, \top) \), which simplifies to \( \top \), meaning that the type inference succeeds everywhere. Next, we combine unifiers with the function \( \text{merge} \) defined earlier in this subsection. Note, since \( \theta_l \) is \( \emptyset \), the second case of \( \text{merge} \) will handle each mapping in \( \theta_r \). For example, as \( \alpha \mapsto \text{Bool} \rightarrow \text{Int} \in \theta_r \), then the merged substitution includes \( \alpha \mapsto A(\kappa_0, \text{Bool} \rightarrow \text{Int}) \), where \( \kappa_0 \) is a fresh type variable. Here we use a fresh type variable in the first alternative to denote that the first alternative for \( \alpha \) is not constrained yet, allowing future unification with any type, if necessary. Overall, let \( \theta_m \) be the substitution after merging \( \theta_l \) and \( \theta_r \), then
\[ \theta_m = \{ \alpha \mapsto A(\kappa_0, \text{Bool} \rightarrow \text{Int}), \kappa_1 \mapsto A(\kappa_0, \text{Bool}), \kappa_2 \mapsto A(\kappa_{10}, \text{Int}), \kappa_4 \mapsto A(\kappa_{12}, \text{Int}) \} \]

Substituting the result type \( A(*, \kappa_2) \rightarrow \text{Int} \) with \( \theta_m \) yields the type \( A(*, A(\kappa_0, \text{Bool} \rightarrow \text{Int})) \rightarrow \text{Int} \), which simplifies to the type \( A(*, \text{Bool} \rightarrow \text{Int}) \rightarrow \text{Int} \)

after we eliminate the unreachable alternative \( \kappa_0 \). Since the combined typing pattern is \( \top \) and selecting \( \top \) with \( \{A.2\} \) yields \( \top \), it means that we can migrate \( x \), the parameter associated with the choice \( A \). Moreover, based on the result type of \( A(*, \text{Bool} \rightarrow \text{Int}) \rightarrow \text{Int} \), we know the migrated expression has the type \( (\text{Bool} \rightarrow \text{Int}) \rightarrow \text{Int} \).

Now we solve the constraint \( A(e, C_1 \land C_2 \land C_4 \land C_5 \land C_6) \) generated for the expression \( \lambda x. * x(\text{succ}(x \text{True})) \). We proceed similarly as before. In particular, constraint solving \( C_1 \) through \( C_4 \) yields the unifier \( \theta_r \) mentioned above. We then need to solve \( C_5 \) and \( C_6 \) from \( \theta_r \). When solving \( C_6 \), we need to unify \( \text{Bool} \rightarrow \text{Int} \) with \( \text{Int} \rightarrow \kappa_6 \), which fails. The pattern returned is thus \( \bot \). Therefore, the pattern for solving the whole constraint is \( A(\top, \bot) \). Based on the pattern we know that we can not migrate \( x \).

Note, even though our approach can not migrate \( x \), types more precise than \( * \) could actually be assigned to \( x \), such as \( * \rightarrow \text{Int} \). The reason we cannot find this migration is that \( \lambda x. x(\text{succ}(x \text{True})) \) is not well-typed under type inference by Garcia & Cimini (2015), and our type inference can be considered as the variational version of theirs. We provide an extension to the unification algorithm \( \mathcal{U} \) to infer more precise types in Section 9.2.

### 7.3 Properties

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

*Theorem 13 (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 \( M_1 \approx M_2 \) only. In fact, we can transform other forms into this form. For example, \( d(M_{11} \approx M_{12}, M_{21} \approx M_{22}) \) can be transformed into \( d(M_{11}, M_{21}) \approx d(M_{12}, M_{22}) \). Note that \( \pi \) in the constraint is just a placeholder and will be updated when the constraint solving finishes.

*Theorem 14 (Soundness)*
Migrating Gradual Types

If $\Upsilon(M_1 \approx M_2) = (\theta, \pi')$, then $\theta(M_1) \approx_{\pi'} \theta(M_2)$.

**Theorem 15 (Completeness)**

Given $M_1 \approx M_2$, if $\theta_1(M_1) \approx_{\pi_1} \theta_1(M_2)$, then $\Upsilon(M_1 \approx M_2) = (\theta_2, \pi_2)$ such that $\pi_1 \leq \pi_2$ and $\theta_1 = \theta \circ \theta_2$ for some $\theta$.

The idea of the proof is to go through all possible constructs of the type $M$ and show that $\Upsilon$ covers all possibilities. To establish that most general unifiers exist, we get the results directly from the induction hypothesis (and compose the mgus of the subterms) or use proof by contradiction. As the proof is standard and lengthy, we omit it here.

### 8 Introducing Dynamism for Fixing Static Type Errors

Fixing static type errors by introducing $\star$s could be useful under several scenarios. First, when migrating a program, the user may have added static types that cause type errors. To pass static type checking of gradual typing, some added type annotations should be removed. Second, the addition of dynamic types can be used to silence type errors and defer the reporting of type errors to runtime (Bayne et al., 2011; Vytiniotis et al., 2012).

This idea is particularly intriguing for fixing static type errors as type error messages generated by compilers are often opaque and difficult to understand (Loncaric et al., 2016; Serrano & Hage, 2016; Munson & Schilling, 2016; Pavlinovic et al., 2014; Marceau et al., 2011a,b). For example, the work by Bayne et al. (2011) shows that obtaining even partial result of ill typed programs helps programmers to understand type errors and accelerate program development. Our recent work indicates that gradual typing leads to more concrete feedback than deferred type errors for ill typed programs (Chen & Campora III, 2019).

In particular, in some situations while deferred type errors dump compile-time error messages, gradual typing returns values to the programmer.

A simple approach for removing type errors is adding $\star$ annotations to all parameters, which are static by default. However, this approach is undesirable for several reasons. First, adding a $\star$ annotation to every single parameter is laborious to programmers. Second, adding all $\star$s hurts the efforts of migrating programs to be static. Third, the program is likely to lose useful type information in many locations.

For this reason, our goal here is to develop a solution to question Q2. Specifically, for a statically ill typed program, we aim to find a minimum set of parameters such that replacing them with $\star$s removes the type error. It turns out that introducing as few dynamic types as possible for answering Q2 is equally tricky as removing as many dynamic types as possible.

To illustrate, consider the following program `rowAt1St`, which shares the body with `rowAt1` but removes $\star$s from all its parameters.

```haskell
rowAt1St headOrFoot fixed widthFunc table border i =
  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
```
This function is ill typed since, for example, the then-branch for computing \( \text{width} \) requires \( \text{widthFunc} \) to have the type \( \text{Bool} \rightarrow \text{Int} \) and the else-branch requires it to have the type \( \text{Int} \rightarrow \text{Int} \).

The difficulties in adding \( \ast \)s are similar to the ones espoused for removing \( \ast \)s in Section 1.1. There is an exponential number of ways \( \ast \)s can be added to the program; adding \( \ast \)s to all parameters introduces more dynamism than desired. Some dynamism can be avoided by adding \( \ast \) annotations in a left to right manner, but this is inefficient and can still add unnecessary dynamism. For example, following this process on \( \text{rowAtISt} \) leads to a migration that add \( \ast \)s from \( \text{headOrFoot} \) to \( \text{border} \), since only then \( \text{rowAtISt} \) becomes well typed. In fact, however, the dynamism on, for example, \( \text{table} \) is unnecessary. If the programmer wants to remove such unnecessary dynamism, they encounter the exact same difficulties detailed in Section 1.1. The similarity in difficulties inspires our solution to introducing dynamism, which is detailed in the next subsection.

### 8.1 Duality to Removing Dynamism

The program \( \text{rowAtISt} \) can be thought of as one of the programs in the migration space of \( \text{rowAtI} \) in Figure 1. In fact, it is the bottom-most program in the figure had we listed out the full migration space there. Recall that programs 3 and 5 were the most static migrations for program 1. While introducing \( \ast \)s for \( \text{rowAtISt} \), programs 3 and 5 are likewise the programs we desire since they keep as many static types as possible and are still well typed.

We can envision organizing the whole migration space into a lattice where more dynamic programs are in the upper portions of the lattice (Takikawa et al., 2016). The process of removing dynamism to make the program static keeps going down the lattice before a type error appears. The process of introducing dynamism to fix type errors keeps going up the lattice until type errors disappear. Overall, these two processes are dual. This fact inspires our formal development to realize the process of introducing dynamism, which we shall see next.

**Typing rules** In removing dynamism, we introduce variations for parameters whose type annotations are \( \ast \)s and not to others. Based on the duality, we should now introduce variations to parameters without \( \ast \) annotations and not to others. Specifically, we define a new type system using the judgment form \( \pi; \Gamma \vdash_D e : M \mid \Omega \). This judgment has the same meaning as the one in Figure 10 and shares the same rules as that one except for \( \text{Abs} \) and \( \text{AbsDyn} \), for which typing rules are as follows.

\[
\begin{align*}
\text{Abs} & \quad \pi; \Gamma, x \mapsto d(\ast, V) \vdash_D e : M \mid \Omega \quad d \text{ fresh} \\
\pi; \Gamma \vdash_D \lambda x. e : d(\ast, V) \rightarrow M \mid \Omega \cup \{ x \mapsto d(\ast, V) \}
\end{align*}
\]

\[
\begin{align*}
\text{AbsDyn} & \quad \pi; \Gamma, x \mapsto \ast \vdash_D e : M \mid \Omega \\
\pi; \Gamma \vdash_D \lambda x : \ast. e : \ast \rightarrow M \mid \Omega
\end{align*}
\]

These two rules are dual to the corresponding ones in Figure 10. For an abstraction with a static type, the type error may be removed by changing its parameter to have the dynamic type. We express this by creating a fresh variation with its first alternative being \( \ast \), as can
be seen in the Abs rule. The rule then records the changes in the variational statifier. For
AbsDyn, no changes will be made for the parameter type, and thus no variations are created
in the rule, since our goal is to fix static type errors and not to migrate programs towards
using more static typing.

Using the given typing rules, we can derive the following type for rowAtIst, assuming
the variation names for parameters from left to right are A, B, D, E, F, G.

\[ A(\star, \text{Bool}) \rightarrow B(\star, \text{Bool}) \rightarrow D(\star, (\text{Int} \rightarrow \text{Int})) \rightarrow E(\star, \text{Char}) \rightarrow F(\star, \alpha) \rightarrow G(\star, \text{Int}) \rightarrow \text{Char} \]

The typing pattern for it is:

\[ \pi_d = B(F(\top, \bot), D(F(\top, \bot), \bot)) \]

**Connection to ITGL**. Each variational statifier (in this context perhaps it should be
renamed to dynamifier) generated by the \( \vdash_D \) type system now collects parameters for which
\( \star \) annotations are added (instead of removed as was done previously). From the variational
statifier, we can generate a statifier for each given decision as follows.

\[ \Omega[\delta] = \{ x \mapsto |M|_\delta \mid x \mapsto M \in \Omega \} \]

The generated statifier coerces certain parameters to have type \( \star \)s and leaves others to
their original types. We can define a type system similar to the type system in Figure 4 that
types gradual expressions under updates from statifiers. The new type system is the same
as the one in Figure 4 except for the rules Abs and AbsDyn, which are presented below.

\[
\begin{align*}
\text{Abs} & : \omega; \Gamma, x \mapsto \omega(x) \vdash_{GCD} e : G \\
\text{AbsDyn} & : \omega; \Gamma, x \mapsto * \vdash_{GCD} e : G
\end{align*}
\]

In Abs, a parameter with a static type is maybe assigned a * if the \( \omega \) specifies so. For
functions with * parameters, handled by AbsDyn, the typing rule does not update their
types.

**Finding error fixes**. The \( \vdash_D \) typing relation indeed finds correct and complete fixes to type
errors, as captured in the following theorems, which serve a similar goal as Theorems 4
through 6 served in the type system of removing dynamism. The proofs of these theorems
thus follow those closely and are omitted here.

**Theorem 16 (Error Fixing Soundness)**

Given \( e \) and \( \Gamma \) assume \( e \) cannot be typed in ITGL under \( \Gamma \). Let \( \pi; \Gamma \vdash_D e : M \mid \Omega \). If
\( |\pi|_\delta = \top \), then \( \Omega[\delta]; \Gamma \vdash_{GCD} e : G \) for some type \( G \).

**Theorem 17 (Error Fixing Completeness)**

If \( \omega; \Gamma \vdash_{GCD} e : G \), then there exists some typing \( \pi; \Gamma \vdash_D e : M \mid \Omega \) where \( |M|_\delta = G \) and \( \Omega[\delta] \)
for some decision \( \delta \).

The previous theorem indicates that we can use migrational typing to fix errors but does
not state that the fixes are minimal. The following theorem states that we can find a most
general, least dynamic fix for a program. We call this the MGDM typing.

**Theorem 18 (Existence of the MGDM typing)**
Given any \( e \) and \( \Gamma \), there is a MGDM typing \( \pi;\Gamma \vdash_D e : M \mid \Omega \) such that for any \( \pi;\Gamma \vdash_D e : M \mid \Omega \), we have \( \forall \delta. [\pi]_\delta = \top \Rightarrow [\pi]_\delta = \top \land [M]_\delta \preceq [M]_\delta \).

From the typing pattern \( \pi \) in MGDM, we can reuse the machinery to find the best migration in Section 5.2 for finding migrations that fix type errors by introducing fewest \( \star \)s to parameters. For example, the \( \pi \) for the MGDM of \texttt{rowAt1st} is \( \pi_d \) given earlier. This pattern indicates that either \texttt{fixed} and \texttt{border} should have \( \star \)s to remove the type error, or \texttt{widthFunc} and \texttt{border} should have \( \star \)s.

### 8.2 Discussion

This section demonstrates that migrational typing is flexible and can be easily adapted to solve another interesting program migration problem. The fundamental reason is that migrational typing provides an efficient method to explore the typing of the full migration space and extract the desired migrations from that space, which naturally lends itself to solving other migration problems.

It is interesting to see if we can fix type errors and migrate programs to utilizing more static typing simultaneously. Essentially, such a process first adds \( \star \) annotations to remove the type error and then inspects to see if other \( \star \) annotations can be safely removed after the error is fixed. Note that typing rules in Figure 10 introduce variations for parameters with \( \star \)s and those in this section introduce variations for parameters that have no \( \star \)s. This suggests that the type system that simultaneously fixes type errors and migrates programs should create variations for all parameters. Specifically, the \( \text{absDyn} \) rule should be the same as the one in Figure 10 while \( \text{abs} \) be the same to the one in \( \vdash_D \). After that, we can use the method described in Section 5.2 to extract the migration that removes type errors as well as migrate the program to be as static as possible.

The simplicity of the type system for this purpose echoes our early observation about the flexibility and adaptability of migrational typing.

### 9 Extensions

In this section, we consider how to support additional language features in our migrational type system. First, we show that our migrational type system is flexible and can support extensions that make the source language more expressive for programmers. Then, we cover other uses of migrational typing, for example allowing programmers to indicate which regions they want to remain dynamic or static.

#### 9.1 Other Language Features

Our version of ITGL, given in Figure 10, restricts parameters to be either unannotated or annotated by \( \star \). The formulation of gradual typing by Garcia & 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 \( G = \star \) or \( G \) is fully static, type the abstraction as usual; if
Migrating Gradual Types

G is a complex type containing * 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 = Int → * → *, then the type of the parameter is d | Int → * → *, Int → V_1 → V_2, where
d is fresh. To generate the corresponding constraint (Section 6), we replace V_1 and V_2 by
fresh type variables. Note that this extension still tries to assign full static types for *s. As
such, this extension will not be find a migration for \( \lambda x: \lambda \alpha \alpha (\text{succ} (x \text{True})) \), as shown in
Section 1.3. The extension in Section 9.2 is able to infer partial static types.

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

\[
\pi; \Gamma \vdash e : M | \Omega \quad G \approx \pi V \quad M \approx \pi d \langle G, V \rangle
\]

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 * types
removed. We can update the the structure of \( \Omega \) 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 \( \alpha \) to denote a list of type variables and \( \{ \alpha \rightarrow V \} \)
to denote a set that includes \( \alpha_1 \rightarrow V_1, \ldots, \alpha_n \rightarrow V_n \). The function \( \text{vars}(\cdot) \) returns the free
type variables in its 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).

\[
\pi; \Gamma \vdash e_1 : M_1 | \Omega_1 \quad \alpha = \text{vars}(M_1) - \text{vars}(\Gamma)
\]

\[
\pi; \Gamma, x \mapsto \forall \alpha. M \vdash e_2 : M_2 | \Omega_2
\]

\[
\pi; \Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : M_2 | \Omega_1 \cup \Omega_2
\]

\[
\text{VAR } x \mapsto \forall \alpha. M \in \Gamma
\]

\[
\pi; \Gamma \vdash x : \{ \alpha \rightarrow V \} (M) | \emptyset
\]

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.

9.2 Inferring More Precise Types

The example in Section 7.1 shows that our approach fails to find a migration for the
expression \( \lambda x: \lambda \alpha \alpha (\text{succ} (x \text{True})) \), even though \( \lambda x: \lambda \alpha \alpha (\text{succ} (x \text{True})) \) can be
a more precise migration. Recall from Section 6 that during constraint generation we
assigned the variational type \( \lambda (\star, \alpha) \) to the parameter type x and the generated constraint
is \( \lambda (e, C_1 \land C_2 \land C_4 \land C_5 \land C_6) \).

To investigate why our approach can not find a migration and how we can potentially
improve this situation, we list the constraint solving process for the constraint \( C_1 \land C_2 \land
C_4 \land C_5 \land C_6 \) below. The first column lists the constraint being solved and the latter two
columns list the unifier and pattern from solving the constraint.
John Peter Campora III, Sheng Chen, Martin Erwig, and Eric Walkingshaw

The main reason our approach fails to find a migration is that, as we were solving the first constraint \( \alpha \approx^? \text{Int} \to \text{K}_3 \), we made three requirements: 1) the type that \( \alpha \) maps to is constructed by the \( \to \) type constructor, 2) the parameter type of \( \to \) be a static type, and 3) the return type of \( \to \) be a static type. However, in \( x(\text{succ} (x \text{True})) \), the body of the function, \( x \) is used as functions and applied to both \( \text{Bool} \) and \( \text{Int} \) values. As a result, no static type could be assigned to \( x \). We can address this problem by relaxing the three requirements for \( \alpha \). To address this problem, we observe that \( \alpha \) denotes the type for \( x \) when the \( * \) for \( x \) is removed, and we are finding a more precise migration than \( * \). Thus, instead of constraining \( \alpha \) with all the three requirements at once, we can relax the latter two requirements and require \( \alpha \) be unified with a type whose type constructor is \( \to \) only. From now on, we call type variables that are introduced to replace \( *s \) for dynamic parameters migration type variables. Migration type variables appear in the right alternatives of choices when choices are first created. We will use \( \alpha \) to range over migration type variables.

Overall, the idea of our solution is that when a migration variable is unified against a function type, we require only that the migration variable be mapped to a function type but allow the parameter type and return type to remain a \( * \). The typing that happens later decides whether the parameter type and/or return type could be made precise than a \( * \). As a result, a parameter can now be migrated to a function type whose parameter or return type remains a \( * \).

One technical challenge is that for the parameter type and return type, we need to explore two possibilities: the \( * \) and a more precise type. Our machinery with variational typing provides a nice solution. Specifically, when a migration variable \( \alpha \) is unified with a function type \( M_1 \to M_2 \), we refine \( \alpha \) to a function type \( A_1(\alpha_1, \alpha_2) \to A_2(\alpha_1, \alpha_2) \) (We refer to this process as refinement) and unify this function type against \( M_1 \to M_2 \). Here, \( A_1, \alpha_1, A_2, \) and \( \alpha_2 \) are fresh and \( \alpha_1 \) and \( \alpha_2 \) are migration variables, which could be further refined to function types whose parameter and return types are \( *s \). The function type \( A_1(\alpha_1, \alpha_2) \to A_2(\alpha_1, \alpha_2) \) encodes four possibilities: both the parameter type and the return type could be \( * \) or a more precise type.

Following this idea, the constraint solving process for the constraints \( C_1 \) through \( C_7 \) is updated to the following. In the “Solution” column below, we omitted the mappings \( \alpha_1 \mapsto \text{K}_1 \) and \( \alpha_2 \mapsto \text{K}_2 \) to save space.
Migrating Gradual Types

(bR) \( \psi (\beta \approx^7 M) \)
\[ \frac{\beta \notin \text{vars}(M) \land \neg \text{hasDyn}(M) = \{ \beta \mapsto M \}, \top}{d \in \text{choices}(M) = \psi(d, \beta, \beta \approx^7 M)} \]
\[ \frac{\beta \notin \text{vars}(M) \land M \text{ is of form } M_1 \to M_2 =}{\exists (\theta_1, \pi_1) = \psi(\beta \approx^7 k_1 \to k_2); (\theta_2, \pi_2) = \psi(k_1 \to k_2 \approx^7 M_1 \to M_2) \text{ in } (\theta_2 \circ \theta_1, \pi_2 \cap \pi_1)} \]
\[ \text{otherwise } = (\emptyset, \bot) \]

(bR*) \( \psi (M \approx^7 \beta) = \psi(\beta \approx^7 M) \)
(b1) \( \psi(\alpha \approx^7 \alpha) = (\emptyset, \top) \)
(b2) \( \psi(\alpha \approx^7 \gamma) = (\{ \alpha \mapsto \gamma \}, \top) \)
(b3) \( \psi(\alpha \approx^7 \beta) = (\{ \alpha \mapsto \beta \}, \top) \)
(b4) \( \psi(\alpha \approx^7 d \ M_1, M_2) = \psi(d, \alpha, \alpha \approx^7 d \ M_1, M_2) \)
(b5) \( \psi(\alpha \approx^7 M_1 \to M_2) \)
\[ \text{AllLvsDynMvs}(M_1 \to M_2) \land \alpha \in \text{vars}(M_1 \to M_2) = (\emptyset, \bot) \]
\[ \text{AllLvsDynMvs}(M_1 \to M_2) \land \neg \text{hasDyn} (M_1 \to M_2) = (\{ \alpha \mapsto M_1 \to M_2 \}, \top) \]
\[ \text{AllLvsDynMvs}(M_1 \to M_2) = \]
\[ \exists (\theta_1, \pi_1) = \psi(\beta \approx^7 k_1 \to k_2); (\theta_2, \pi_2) = \psi(k_1 \to k_2 \approx^7 M_1 \to M_2) \text{ in } (\theta_2 \circ \theta_1, \pi_2 \cap \pi_1) \]
\[ \text{otherwise } = \]
\[ \text{let } \theta_1 = \{ \alpha \mapsto A_1(\ast, \alpha_1) \to A_2(\ast, \alpha_2) \} \]
\[ \text{A}_1, A_2, \alpha_1, \text{ and } \alpha_2 \text{ fresh} \]
\[ (\theta_2, \pi_2) = \psi(A_1(\ast, \alpha_1) \to A_2(\ast, \alpha_2) \approx^7 \theta_1(M_1 \to M_2)) \]
\[ \text{in } (\theta_2 \circ \theta_1, \pi_2) \]
(b6) \( \psi (M \approx^7 \alpha) = \psi(\alpha \approx^7 M) \)

Fig. 14: An extension to the unification algorithm in Figure 13.

---

<table>
<thead>
<tr>
<th>Constraint</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \alpha \approx^7 k_1 \to k_2 )</td>
<td>( { \alpha \mapsto A_1(\ast, k_1) \to A_2(\ast, k_2) } )</td>
</tr>
<tr>
<td>( \alpha \approx^7 \text{Bool} \to k_4 )</td>
<td>( { \alpha \mapsto A_1(\ast, \text{Bool}) \to A_2(\ast, k_4), k_1 \mapsto \text{Bool}, k_2 \mapsto k_4 } )</td>
</tr>
<tr>
<td>( \text{Int} \approx^7 k_2 )</td>
<td>( { \alpha \mapsto A_1(\ast, \text{Int}) \to A_2(\ast, k_2) \to \text{Int} } )</td>
</tr>
<tr>
<td>( \alpha \approx^7 k_5 \to k_6 )</td>
<td>( { \alpha \mapsto A_1(\ast, \text{Int}) \to A_2(\ast, \text{Int}), k_5 \mapsto \text{Bool}, k_6 \mapsto \text{Int} } )</td>
</tr>
</tbody>
</table>

From Section 6 (page 32), we know that the type of \( \lambda x : \ast . x(\text{succ}(x \text{True})) \) is \( \ast \)

A(\ast, \alpha) \to A(\ast, k_0). Plugging in the solution for \( \alpha \) from the unifier above, the type for \( \lambda x : \ast . x(\text{succ}(x \text{True})) \) is \( M_{dp} = A(\ast, A_1(\ast, \text{Bool}) \to A_2(\ast, \text{Int})) \to A(\ast, A_2(\ast, k_0, \text{Int})) \).

Moreover, the pattern for the whole function is \( A(\top, A_1(\top, \bot)) \). Note, \( A_2 \) does not appear in the result pattern because whether we choose \( \ast \) or \( \text{Int} \) for the return type of the function type for \( \alpha \), the well-typedness of the expression remains the same. Applying the operations ve and expand, defined in Section 5.2, to the pattern \( A(\top, A_1(\top, \bot)) \), we know that the best migration for this expression corresponds to the valid eliminator \( \{ A_1, A_2 \} \). Selecting \( M_{dp} \) with \( \{ A_1, A_2 \} \) yields the type \( (\ast \to \text{Int}) \to \text{Int} \), the type of \( \lambda x : \ast . x(\text{succ}(x \text{True})) \) after migrating the parameter \( x \). This means that our extension could indeed find a more precise migration for \( \lambda x : \ast . x(\text{succ}(x \text{True})) \).

An extension to the unification algorithm Figure 14 presents an extension to the unification algorithm that implements our idea from above. We briefly go through the cases.

First, the cases (bR) and (bR*) replace cases (b) and (b*) in Figure 13, by renaming the type variables \( \alpha \) to \( \beta \). Note that from now on, we use \( \alpha \) to denote migration variables and \( \beta \) to denote all other variables. The cases (b1) through (b4) handle unification between...
a migration variable and itself, a constant type, a non-migration type variable, and a variational type.

Case (b5) handles the unification between a migration variable and a function type. This case uses an auxiliary function $\text{AllLvsDynMvs}$ to determine if the leaves of a given input type are all *s or migration type variables. For example, all $\text{AllLvsDynMvs}(\alpha_1 \rightarrow \alpha)$, $\text{AllLvsDynMvs}(\alpha_2)$, and $\text{AllLvsDynMvs}(\langle \alpha \rightarrow \alpha \rangle \rightarrow \alpha_2)$ are true, while $\text{AllLvsDynMvs}(\alpha_1 \rightarrow \text{Int})$ and $\text{AllLvsDynMvs}(\langle \alpha \rightarrow \text{Bool} \rangle \rightarrow \alpha_2)$ are false. This function helps avoid non-termination in our extension. To illustrate, consider the constraint $\alpha \approx ? \alpha \rightarrow \beta$. Such a constraint arises when typing a self application, such as in the expression $\lambda \text{x}. \star \text{x} \text{x}$. This constraint fails to solve using the constraint solving algorithm in Figure 13 due to the occurs check.

With the extension in Figure 14, we will turn the constraint $\alpha \approx ? \alpha \rightarrow \beta$ into $A_1(\star, \alpha_1) \rightarrow A_2(\star, \alpha_2) \approx ? (A_1(\star, \alpha_1) \rightarrow A_2(\star, \alpha_2)) \rightarrow \beta$. This constraint encodes four constraints, and one of them is $\alpha_1 \rightarrow \alpha_2 \approx ? (\alpha_1 \rightarrow \alpha_2) \rightarrow \beta$ (if we select the variational constraint with the decision $\{A_1, 2, A_2, 2\}$). We observe that this problem is larger than the original problem $\alpha \approx ? \alpha \rightarrow \beta$ and the constraint between the parameter types ($\alpha_1 \approx ? \alpha_2$) resembles the original problem. We can envision that the unification will not terminate if we keep on refining migration variables as we did above.

There are two potential ways to address this problem. The first is that we use a heuristic, such as allowing a single migration variable be refined by up to a certain number of times only. Any further refinement attempt on the same migration variable would be rejected and treated as a unification failure. The second is to detect the unification that unifies a migration variable ($\alpha$) against a function type that contains the migration variable ($\alpha$) and all other leaves are other migration variables or *s. Such a unification does not reflect any program structure information but is resulted from refining a migration variable to a function type, since constraint generation (Figure 11) does not generate such a constraint. If such a unification problem is detected, we can terminate the unification with a failure.

Note, even though unification will fail for $\alpha_1 \rightarrow \alpha_2 \approx ? (\alpha_1 \rightarrow \alpha_2) \rightarrow \beta$, which means the typing pattern returned for unifying it will be $\bot$, the typing pattern for unifying $A_1(\star, \alpha_1) \rightarrow A_2(\star, \alpha_2) \approx ? (A_1(\star, \alpha_1) \rightarrow A_2(\star, \alpha_2)) \rightarrow \beta$ will not be $\bot$. It is $A_1(\top, \bot)$. This means that the pattern for solving $\alpha \approx ? \alpha \rightarrow \beta$ is not $\bot$.

In this extension, we use the second way to address this problem. Concretely, we capture it in the first subcase of case (b5). In the second subcase, $\alpha$ does not occur in the function type and all leaves are migration variables, then we directly map $\alpha$ to the function type.

In the third subcase, the function type contains some *s. We need to refine $\alpha$ to a function type, but without creating new variations. The last subcase implements the idea of refining a migration variable into a function type whose both parameter and return types are variations.

With this extension, let’s now turn to finding migrations for the term $\lambda \text{x}. \star \text{x} \text{x}$. First, we generate the constraint $A(\star, \alpha) \approx ? A(\star, \alpha) \rightarrow \beta$ and the type for the term is $A(\star, \alpha) \rightarrow \beta$. This constraint will be solved using case (d) of Figure 13, which will solve two constraints originated from the two alternatives of $A$. For the left alternative, the constraint is $\star \approx ? \star \rightarrow \beta$, which will be solved by case (a) of Figure 13 with the solution $\langle \beta, \top \rangle$. For the right alternative, the constraint is $\alpha \approx ? \alpha \rightarrow \beta$. This constraint will be handled by the fourth
Migrating Gradual Types

...subcase of case (b5) in Figure 14, and it will be transformed to $A_1(\star, \alpha_1) \rightarrow A_2(\star, \alpha_2) \approx^2$

$(A_1(\star, \alpha_1) \rightarrow A_2(\star, \alpha_2)) \rightarrow \beta$.

With a few steps, this problem can be solved and the solution is $\{\alpha \rightarrow A_1(\star, \alpha_1) \rightarrow A_2(\star, \beta), \alpha_2 \rightarrow \beta\}$ and the pattern is $A_1(\top, \bot)$. Substituting the type of the term with this solution yields $A(\star, A_1(\star, \alpha_1) \rightarrow A_2(\star, \beta)) \rightarrow \beta$ and the overall pattern is $A(\top, A_1(\top, \bot))$. From this pattern, we can use $ve$ and expand defined in Section 5.2 to calculate the strictest valid eliminator $A(2, A_1, A_2, 2)$. Selecting the type $A(\star, A_1(\star, \alpha_1) \rightarrow A_2(\star, \beta)) \rightarrow \beta$ with this eliminator leads to the type $(\star \rightarrow \beta) \rightarrow \beta$, which is a most static migration for $\lambda x: \star. x$. This shows that with the extended constraint solving algorithm, we could find a more precise migration for $\lambda x: \star. x$ that we could not find earlier.

9.3 Further Migration Scenarios

Sections 4 and 5 provide a type system and a method for finding all best migrations. In practice, there may be different migration requirements. In this subsection, we explore a few of them and show how to support them with machinery developed in earlier sections. Specifically, we consider the following migration scenarios.

(i) Can the programmer control which parameters must or must not be migrated?

(ii) If migrating a set of indicated parameters yields a type error, can we still migrate a subset of these parameters?

(iii) Given a set of parameters, can we find which parameters cannot be migrated in unison?

(iv) Can we find the migrations that migrate the greatest number of parameters?

We use the program $rowAtI$ to illustrate these scenarios and the development of corresponding machinery. Recall that the variations introduced for the parameters $fixed, widthFunc, table, border$, and $i$ are $A, B, D, E, and F$, respectively. The typing pattern for this program was shown in Section 4.5 and is reproduced here for readability.

$$\pi_0 = A(E(T, \bot), B(E(T, \bot), \bot))$$

We next go through each scenario.

Scenario (i): We begin with a concrete case. Assume that the programmer requires that $table$ must be migrated and $widthFunc$ must not be migrated. We can build a decision $\delta_r$ for refining the pattern $\pi_a$ based on this requirement. To express that $table$ must be migrated, we extend $\delta_r$ with $D.2$, as $D$ is the variation introduced for $table$. For $widthFunc$ to be not migrated, we extend $\delta_r$ with $B.1$, making $\delta_r = \{B.1, D.2\}$. After that, we refine $\pi_a$ with $\delta_r$, yielding the new pattern $A/E(T, \bot), E(T, \bot)$, which could be simplified to $E(T, \bot)$. We can now apply the method developed in Section 5 to the pattern $E(T, \bot)$ to find the best migrations for $\piAtW$ while honoring the requirements. Based on the pattern $E(T, \bot)$, the migration result is that $border$, the parameter corresponds to $E$, can not be migrated, and all other parameters can be migrated. Overall, the migration is that we can migrate $fixed, i$, and $table$.

In general, for a program and its typing pattern $\pi$ generated from MGSM, we follow the following steps to handle this scenario.
(1) For each parameter that must be migrated, we extend $\delta_i$ with $d.2$, where $d$ is the
variation introduced for the parameter.

(2) For each parameter that must not be migrated, we extend $\delta_i$ with $d.1$, where $d$ is the
variation introduced for the parameter.

(3) We refine the pattern $\pi$ with $\delta_i$.

(4) With the resulting pattern from the last step, we use the method for finding most
static migrations outlined in Section 5.2 to find desired migrations.

Scenario (ii): Assume that the programmer requires to migrate all $\text{fixed}$, $\text{widthFunc}$,
and $\text{table}$. According to the process of calculating $\delta_i$ given earlier, $\delta_i = \{A.2,B.2,D.2\}$.
We observe that $[\pi_0]_{\delta_i} = \bot$, indicating that not all these parameters can be migrated at the
same time. However, the $\bot$ does not indicate that none of the parameters can be migrated.

To figure out if a parameter within the specified set could be migrated, we could list
all decisions yielding best migrations and check if the parameter appears in any set.
For example, based on Section 5.2, the decisions corresponding to best migrations for
$\text{rowAtI}$ are $\{A.2,B.1,D.2,E.1,F.2\}$ and $\{A.1,B.2,D.2,E.1,F.2\}$. From the first set, we
could decide that $\text{fixed}$ (since $\text{fixed}$ corresponds to $A$ and $A.2$ belongs to the set) and
$\text{table}$ of the desired set could be migrated. From the second set, we could decide that
$\text{widthFunc}$ and $\text{table}$ could be migrated. In this case, we have two different such sets. In
other cases, we may have only one such set. For example, if the programmer indicated that
they wanted to migrate $\text{fixed}$ and $\text{border}$, then the unique migration corresponds to the
decision is $\{A.2,B.1,D.2,E.1,F.2\}$, indicating that only $\text{fixed}$ within the two parameters
could be migrated.

Scenario (iii): During program migration, it is quite common that migrating one
parameter may preclude the migration of others. For example, in $\text{rowAtI}$, we could not
migrate $\text{widthFunc}$ if we have migrated $\text{fixed}$ and vice versa. Therefore, presenting the
unison parameters that could no longer be migrated can be useful to programmers.

Assume that the programmer has migrated $\text{fixed}$ and that we want to calculate the
impact it has on other parameters. We must now consider two cases. The first case migrates
$\text{fixed}$, and the decision is $\delta_i = \{A.2\}$. The second case does not migrate $\text{fixed}$, and the
decision is $\delta_{\neg r} = \{A.1\}$. Let $\pi_r$ and $\pi_{\neg r}$ denote the typing patterns resulted from selecting
$\pi_0$ with $\delta_i$ and $\delta_{\neg r}$, respectively, we have

$$\pi_r = B\{E(\top, \bot), \bot\} \quad \pi_{\neg r} = E(\top, \bot)$$

In the first case, from $\pi_r$, we have two decisions that lead to $\bot$: $\{B.1,E.2\}$ and $\{B.2\}$.
In the second case, from $\pi_{\neg r}$, only one decision leads to $\bot$: $\{E.2\}$. By comparing the
decisions in these two cases, we observe that both cases contain $E.2$. This implies that
migrating $\text{border}$, the parameter corresponding to $E$, always causes an error, meaning that
$\text{fixed}$ being migrated was irrelevant to the reason $\text{border}$ cannot be migrated. On the other
hand, only a decision in the first case contains $B.2$ while none in the second case contains it.
This implies that the reason $\text{widthFunc}$ can not be migrated is because $\text{fixed}$ was migrated.
Consequently, the parameter that can not be migrated in unison with $\text{fixed}$ is $\text{widthFunc}$.

Given an expression $e$ and $\pi$ for its MGSM typing, and assume the parameter $x$ is
migrated and the introduced variation for $x$ is $d$, the following steps list the process of
finding parameters that can not be migrated due to the migration of $x$. 

John Peter Campora III, Sheng Chen, Martin Erwig, and Eric Walkingshaw
Migrating Gradual Types

1. Let $\pi_r = \lfloor \pi \rfloor_{d.1}$ and $\pi_{\neg r} = \lfloor \pi \rfloor_{d.2}$.
2. Collect the decisions that produce $\bot$ when selecting $\pi$ with $\pi_r$.
3. Collect the decisions that produce $\bot$ when selecting $\pi$ with $\pi_{\neg r}$.
4. For any $d'$, if $d'.2$ appears in some decisions from step (3) but not from any of the decision in step (2), then the parameter that corresponds to $d'$ cannot be migrated in unison with $x$.

Scenario (iv): This scenario aims to find out the migrations that migrate the greatest number of parameters, which we refer to as maximal migrations. For example, if one static migration migrates two parameters while another migrates four, then the latter is a maximal migration if no other migrations migrate more than four parameters. In some situation, maximal migrations are not unique. For example, two static migrations for $\text{rowAtI}$ migrate three parameters and both are maximal.

Given an expression and its typing pattern $\pi$ for its MGSM, a simple process to find maximal migrations is generate all best migrations from $\pi$ and filter out the migrations that migrate the greatest number of parameters. This process is straightforward and necessitates no changes to our existing machinery, but is computationally expensive. We can improve the efficiency by slightly adapting the $ve$ function for collecting best migrations from Section 5.2. Specifically, for each internal node of the typing pattern, we compare the cardinality of the decisions from the left and right subtrees and discard the decisions that have more left selectors, which are selectors of the form $d.1$ for some $d$ (see Section 2.2). We express this idea in the following function $mve$.

$$mve(\top) = \{\emptyset\}$$
$$mve(\bot) = \emptyset$$
$$mve(d[\pi_1, \pi_2]) = \begin{cases} 
\text{lmve} & \text{rmve} = \emptyset \text{ or } |\mathcal{D}| - |\text{lmve}[0]|_{11} > |\mathcal{D}| - |\text{rmve}[0]|_{11} \\
\text{rmve} & |\mathcal{D}| - |\text{lmve}[0]|_{11} < |\mathcal{D}| - |\text{rmve}[0]|_{11} \\
\text{lmve} \cup \text{rmve} & \text{otherwise} 
\end{cases}$$

where $\text{lmve} = \{\{d.1\} \cup l \mid l \in \text{lmve}(\pi_1)\}$
$$\text{rmve} = \{\{d.2\} \cup r \mid r \in \text{rmve}(\pi_2)\}$$

In the definition, $\delta |_{1}$ (introduced in Section 4.5) returns all left selectors in $\delta$. The notation $\text{lmve}[0]$ returns any member from the set $\text{lmve}$. This is valid because all of the members in $\text{lmve}$ include the same number of left selectors, and so do those in $\text{rmve}$.

The set $\mathcal{D}$ (introduced in Section 5.2) contains all variations introduced in typing $e$. Note, given a decision $\delta$, if $d.1 \notin \delta$ then the parameter corresponding to $d$ can not be migrated. Therefore, $|\mathcal{D}| - |\text{lmve}[0]|_{11}$ gives the number of parameters that can be migrated in $\text{lmve}[0]$.

$mve$ is always more efficient than $ve$ since the former keeps the set of decisions that yield maximal migrations only while the latter keeps all best migrations. In particular, if there is a unique maximal migration, then $mve$ returns only one decision.

Discussion Supporting these scenarios by reusing or slightly adapting existing machinery demonstrates the generality of our approach. We can also support variations or combinations of scenarios we looked at with ease. For example, a combination of
### 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 9.1, the prototype also supports recursive functions, a built-in list type, a built-in `Vector` type, and a tuple type, which are needed to encode the examples described below.

To evaluate the performance of our idea in practice, we have converted programs in Grift to the language supported by our prototype. We used all the programs from Kuhlenschmidt et al. (2019) except the program sieve, which uses recursive types that are not supported in our prototype. Since these converted programs are all well typed, we seed errors in the programs by randomly applying between 2 and 25 changes in each. Each change replaces one leaf of the AST (a variable reference or constant) with another leaf. These programs are summarized in columns 2–5 of Figure 15, showing size in lines of non-blank code, number of functions, number of dynamic parameters, and the number of leaves that were changed.

For each evaluated 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 columns seven through nine of the table. In standard gradual typing, we

---

<table>
<thead>
<tr>
<th>Name</th>
<th>Size</th>
<th># Func</th>
<th># Para</th>
<th># Chg.</th>
<th># Best</th>
<th>Gradual</th>
<th>Brute</th>
<th>Migrational</th>
</tr>
</thead>
<tbody>
<tr>
<td>array</td>
<td>31</td>
<td>5</td>
<td>6</td>
<td>2</td>
<td>1</td>
<td>8.7e-3</td>
<td>0.45</td>
<td>1.9e-2</td>
</tr>
<tr>
<td>blackscholes</td>
<td>125</td>
<td>8</td>
<td>17</td>
<td>10</td>
<td>23</td>
<td>2.1e-2</td>
<td>–</td>
<td>6.7e-2</td>
</tr>
<tr>
<td>fft</td>
<td>93</td>
<td>5</td>
<td>19</td>
<td>2</td>
<td>2</td>
<td>1.9e-2</td>
<td>–</td>
<td>4.4e-2</td>
</tr>
<tr>
<td>matmult</td>
<td>29</td>
<td>3</td>
<td>8</td>
<td>2</td>
<td>1</td>
<td>3.5e-3</td>
<td>0.82</td>
<td>1.1e-2</td>
</tr>
<tr>
<td>nbody</td>
<td>187</td>
<td>21</td>
<td>44</td>
<td>20</td>
<td>31</td>
<td>6.4e-2</td>
<td>–</td>
<td>0.25</td>
</tr>
<tr>
<td>quicksort</td>
<td>44</td>
<td>3</td>
<td>9</td>
<td>2</td>
<td>2</td>
<td>7.8e-3</td>
<td>3.37</td>
<td>2.4e-2</td>
</tr>
<tr>
<td>raytrace</td>
<td>207</td>
<td>20</td>
<td>45</td>
<td>25</td>
<td>46</td>
<td>0.11</td>
<td>–</td>
<td>0.36</td>
</tr>
</tbody>
</table>

Fig. 15: Running time (in seconds) of migrational typing on programs converted from Grift (Kuhlenschmidt et al., 2019). For each row, columns 2 through 4 give the metric of the program, including the number of lines of non-blank code, the number of functions, the number of dynamic parameters, and the number of changes we made to the program. 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.
Migrating Gradual Types

Fig. 16: Relations between ratios of typed parameters and migrational typing times for the nbody benchmark.

run our implementation without creating any variations. 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. In Section 1.1 we discussed that an exploration of all programs are needed to find best migrations. We omit the time for computing the most static migrations from the figure because the time is always within 0.04 seconds.

We observe that the brute-force approach, as expected, is exponentially slower than gradual typing, and it successfully types only the programs that have fewer than 10 parameters. On the other hand, migrational typing scales linearly with the size of the program and exhibits only a 2–3.5 times overhead over gradual typing.

We have also investigated the impact of the ratio of typed parameters on migrational typing time, and we presented the results in Figure 16. Note that the x-axis cuts off at 93% because, as we made random changes to the program, not all parameters can be given static types. In general, a higher ratio of typed parameters leads to fewer variations being created, and thus takes shorter time for migrational typing to finish.

11 Related Work

11.1 Annotation Upgrading and Migratory Typing

Tansey & 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 \( \ast \) 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. In general the focus of this work is about designing such a sister language such that types can be assigned to existing programs in the dynamic language with minimal refactoring. While programmers have to manually add type annotations to make programs more static in migratory typing, migrational typing supports systematically typing the whole migration space and automatically finding the best migrations.

This means that a large focus of migratory typing is orthogonal to our work in that we assume we are working within a given gradual language, and that we do not have to design a static sister language to a dynamic language. On the other hand, if we were given a static language and gradualized it via the idea of Garcia et al. (2016); Cimini & Siek (2016, 2017) we conjecture we could design a migration tool for gradualized languages that supported unification based type inference.

### 11.2 Gradual Typing Migration

As discussed in Section 1.3, this work is closely related to the work by Migeed & Palsberg (2019) on finding maximal migrations for gradual programs. There are several similarities in their work and ours. For example, they consider a set of possible migrations for a gradually typed programs and try to find all of the maximal migrations. These maximal migrations are migrations that cannot add any more type information to the program without causing a static type error, which are similar to our most static migrations. They show that the process of finding maximal migrations is NP hard.

Their work has some notable differences with our work, however. Mainly, the language they consider is a version of GTLC (Siek et al., 2015) with the ability to add \( \texttt{Bool} \) and \( \texttt{Int} \) annotations. In contrast, we start with ITGL, a gradualized version of the Hindley-Milner language, which has a principal type inference that works on unannotated terms. Essentially, while both work aims to find maximal migrations, they use different techniques and criteria. In their work, they continuously generate more precise programs by replacing a \( \ast \) with a more precise type and tests the well-typedness of the generated program. They find a maximal migration if no more \( \ast \)s exist of no more \( \ast \) could be replaced with any more precise type. A migration in our work is maximal if no further \( \ast \) can be eliminated with respect to ITGL Garcia & Cimini (2015) constraint solving. As a result, their approach may find types that are rejected by the ITGL inference that we adapt. For example, for \( \lambda x : \ast . x (\textsf{succ} (x \texttt{True})) \), their approach infers that \( x \) can be given the type \( \ast \rightarrow \texttt{Int} \), whereas our approach respects ITGL, which considers the use of \( x \) to be ill typed (Our extension in Section 9.2 does infer that \( x \) may be migrated to the type \( \ast \rightarrow \texttt{Int} \)).
Finally, we have evaluated the efficiency of our approach on large programs, and we observed that finding all best migrations in our approach is usually within a factor of 2 of typing each possible migration. The efficiency in their approach is unclear. It would be interesting as future work to see if our machinery could be exploited to improve the efficiency of their work.

Phipps-Costin et al. (2021) developed a framework named TypeWhich for migrating gradual types. While both our work and the work by Migeed & Palsberg (2019) aim at maximizing type precision during migration, TypeWhich allows users to consider not only type precision, but also type safety (such that migration does not introduce runtime errors) and type compatibility (such that migration does not break the interoperability between migrated and un-migrated code). As such, some migrations in our work and that by Migeed & Palsberg (2019) may introduce dynamic runtime errors, but not in the safety mode of TypeWhich. The latter two modes are particularly useful because migrations are often not done for the whole project and the migration process should not break code interactions.

In addition, our work and TypeWhich differ in many aspects. First, our work can find all best migrations for a given program whereas TypeWhich finds just one best migration. Consider, for example, the following expression.

\[
\text{width fixed widthFunc = 2 + (if fixed then widthFunc fixed else widthFunc 33)}
\]

TypeWhich displays the following migration for this function when prioritizing type precision.

\[
\text{width (fixed:Int) (widthFunc:Int -> Int) = 2 + (if (fixed:*)) then widthFunc fixed else widthFunc 33)}
\]

Our work finds two best migrations for the function width, and neither is more precise than the other. In the first migration, the type for fixed remains to be * whereas the type for widthFunc is Int-> Int, as shown below.

\[
\text{width (fixed:*)) (widthFunc:Int -> Int) = 2 + (if fixed then widthFunc fixed else widthFunc 33)}
\]

In the second migration, the type for fixed is migrated to Bool and the type for widthFunc is migrated to *-> Int (without the extension in Section 9.2 the type for widthFunc will remain *). The migrated program is shown below.

\[
\text{width (fixed:Bool) (widthFunc:* -> Int) = 2 + (if fixed then widthFunc fixed else widthFunc 33)}
\]

For programs that can not be fully, statically typed, it is likely that hundreds of best migrations exist. Our approach finds all of them in time linear to the size of the program. Since our approach may find a large number of best migrations, it is helpful to allow users to specify preferences about where migrations are preferred. We support them through extensions in Section 9.3. Since TypeWhich finds only one best migration, such supports are not necessary.

Second, by design, TypeWhich may ascribe a * type to a subexpression even though the subexpression has a static type during static type checking. This design allows more parameters to be migrated when precision is maximized. For example, in the migration for width above, TypeWhich ascribed * to fixed that has the type Int so that fixed can be used
where a $\texttt{Bool}$ is needed. Without the ascription, the migrated program is statically ill-typed.

In fact, the migration by TypeWhich will always yield a runtime type error. The migrated $\texttt{width}$ function accepts only $\texttt{Int}$ values, which will lead to a runtime error since $\texttt{fixed}$ is used as a $\texttt{Bool}$ value in the function definition. Our approach does not use ascription for maximizing migrations.

Third, our approach supports polymorphism through let (Section 9.1) while TypeWhich does not. Also, our approach allows programmers to specify type annotations for some parameters and migrations will respect these annotations. In TypeWhich, static type annotations are erased, so that all parameters have the $\star$ types before migration.

Henglein & Rehof (1995) developed an approach for embedding Scheme programs in ML by inserting coercions into subexpressions whose type correctness cannot be statically verified. Their approach uses type inference to reduce coercions that will be inserted. Their approach is similar to TypeWhich that prioritizes type safety.

### 11.3 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 & Tah, 2007), ownership types (Sergey & Clarke, 2012), refinement types (Lehmann & Tanter, 2017; Jafery & Dunfield, 2017; Wadler & Findler, 2009; Williams et al., 2018), session types (Igarashi et al., 2017), and union and intersection types (Castagna & Lanvin, 2017; Castagna et al., 2019; Toro & Tanter, 2017; Siek & Tobin-Hochstadt, 2016). From this perspective, our type system studies the combination of variational types with gradual types. Gradual languages with type inference (Siek & Vachharajani, 2008; Garcia & Cimini, 2015; Rastogi et al., 2012) 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 variations 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 & Findler, 2009, 2007; Tobin-Hochstadt & Felleisen, 2006) adapts the contract system notion of blame so that less precise parts of a program are blamed when cast errors occur. Ahmed et al. (2011, 2017) extended that work to further handle polymorphic types. Since those works, there has been a number of papers involving parametricity in the gradually typed setting (Toro et al., 2019; New et al., 2019). 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.

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 & Siek, 2016, 2017) can produce a gradual type system and dynamic
Migrating Gradual Types

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 7.

11.4 Type inference

The goal of gradual typing is to find out what parameters can be given static types. As
such, gradual typing is closely related to the idea of type inference.

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 flow-based
inference (Garcia & 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 as our approach is not flow-directed.

The inference in Flow (Chaudhuri et al., 2017) is also flow-based and was specifically
designed to not produce false positives for idioms that are commonly used in JavaScript.
It is possible that migrational typing can help the inference process for languages like
JavaScript by using variations to reason about idioms in messy scenarios. A flow-based
inference was also employed over Reticulated Python’s cast inserted transient translation.
The inference was used to optimize program performance, removing unnecessary casts
where the inference indicated that it was safe.

A few type systems, such as Guha et al. (2011); Chugh et al. (2012); Pearce (2013),
support flow-based reasoning but do not perform type inference.

SimTyper, developed by Kazerouian et al. (2021), aims to infer usable types for
Ruby. Unlike most type inference algorithms, the goal of SimTyper is not to infer most
general (precise) types, which could be verbose and hard to use in presence of subtyping,
structural types, overloading, and other dynamic language features. Instead, the goal of
SimTyper is to infer usable types that programmers often write. SimTyper is built on
InferDL, Kazerouian et al. (2020), a heuristics-based type inference algorithm, and a
type equality prediction method based on machine learning. Essentially, when SimTyper
discovers an overly general, complicated type, it uses the type equality predictor to find
a type that is more concise and is equal. SimTyper than uses that more concise type to
replace the complicated one and check if that replacement violates any typing constraint. It
accepts the concise type if no violations detected and rejects the type and look for another
concise type otherwise.

Wei et al. (2020) developed LambdaNet for inferring types for TypeScript. Given a
program, LambdaNet first transforms it to a type dependency graph, where nodes are
type variables for subexpressions in the programs and hyperedges express constraints
(such as the subtyping relation or type equality). Hyperedges may also provide hints to
type inference, such as variables giving rise to the connected type variables have similar
names. All type variables are then converted to vectors of numbers (known as embedding in
machine learning) and, LambdaNet uses a set of rules to propagate type information across
the dependency graphs. These rules manipulate the embedding in each node. As with deep
learning (Neocleous & Schizas, 2002), the intuitions behind such rules are unclear. Finally, after propagation completes, inferred types are readout from embeddings.

11.5 Variational Typing and Others

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 & 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 & 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 $\star$ 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.

The variational cost analysis by Campora et al. (2018b) provided an approach that harmonizes type safety and gradual typing performance. The motivation of that work was that migrating programs will likely slowdown program performance. The solution in that work was constructing a “cost lattice” that estimates the runtime overhead induced by type annotations and comparing costs of different migrations. The solution supports different migration scenarios while adding type annotations, for example finding the migrations that yield the best performance. Technically, that work adapted cost analysis for functional programs (Danner et al., 2015) to a gradually typed language. That work also used the machinery of variational typing to reusing typing and cost analysis to efficiently build the cost lattice.

It is possible that type annotations added by programmers during migrations may cause runtime type errors. Campora & Chen (2020) presented a static type system for detecting runtime type errors, finding out the $\star$s that prevent the runtime type errors from being detected by the static type system, and suggesting fixes to remove dynamic runtime type errors.

Variational typing is defined in terms of the choice calculus (Erwig & Walkingshaw, 2011). Other applications of the choice calculus include the development of variational data structures (Walkingshaw et al., 2014; Meng et al., 2017; Smeltzer & Erwig, 2017) to support variational program execution (Chen et al., 2016; Erwig & Walkingshaw, 2013; Nguyen et al., 2014), and view-based editing of variational programs (Walkingshaw & Ostermann, 2014; Stănciulescu et al., 2016).
Typing patterns in our work have a close resemblance to BDD (Binary Decision Diagrams) of Boolean formulas (Akers, 1978; Bryant, 1992). For example, choices in patterns correspond to internal nodes in BDD, \( \bot \) and \( \top \) correspond to leaves 0 and 1 in BDDs, respectively, and selecting the right alternative of a choice corresponds to following the high edge of an internal node. Moreover, the idea of pattern normal forms, introduced before Theorem 9, are similar to reduced BDDs. Variable ordering has a significant impact on the size of a BDD. The number of nodes of a BDD may be linear to the number of variables under one ordering but it could be exponential under another. Similarly, the ordering of choice names impact the size of a typing pattern. For example, the pattern \( A(\bot, B(\top, C(\bot, \top))) \) has three internal nodes and four leaves, while an equivalent pattern \( C(A(\bot, B(\top, \bot)), A(\bot, \top)) \) has four internal nodes and five leaves.

Due to the reasons below, we conjecture that the ordering problem in our work is not as critical as in BDDs. First, the ordering problem becomes more conspicuous when the leaves mix \( \bot \)s and \( \top \)s. Instead, due to the fact that left alternatives of choices have \( \ast \)s when they are created and \( \ast \)s unify with any types, left subtrees of patterns tend to have \( \top \)s. Section 5.1 gives a formal account of this. For such patterns, the impact of ordering on sizes decreases. For example, \( A(\top, B(\top, C(\top, \bot))) \) has seven nodes, and \( C(\top, A(\top, B(\top, \bot))) \), an equivalent pattern but with different ordering, also has seven nodes. Second, as explained in Section 5.2 (the last second paragraph), typing patterns are usually small, this makes the ordering less important, as even a suboptimal ordering will not cause the pattern to have too many nodes.

12 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 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. It achieves this by conceptually typing 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. Later, we showed that migrational typing can be adapted to statically reason about the number of dynamic casts that will be generated by different points in the migration space so that we can support migration scenarios that consider programmers’ typing goals and performance goals (Campora et al., 2018b). In future work, we plan to see if we can adapt migrational typing to work with a non-unification based inference. This will allow it to analyze gradual languages with object oriented features like Reticulated Python or TypeScript with greater ease. We also plan to explore whether migrational typing can be adapted to provided an analysis of the runtime safety of casts in gradual programs.
Acknowledgments

We thank the anonymous POPL and JFP reviewers and Jens Palsberg for their constructive feedback, which have significantly improved both the content and presentations of this paper. This work was partially supported by the National Science Foundation under the grant CCF-1750886.

References


John Peter Campora III, Sheng Chen, Martin Erwig, and Eric Walkingshaw


van Keeken, Peter. 2006 (October). Analyzing helium programs obtained through logging—the process of mining novice haskell programs. M.Phil. thesis, Department of Information and Computing Sciences, Utrecht University.


A Proofs

This appendix provides proofs to most theorems whose proofs are not given in the paper.

A.1 Proofs of Theorems 1 Through 3

In proving these theorems below, we will make use of two properties about selection on types, expressed in the following lemmas.

Lemma 3 (Selection is idempotent)
For any \( d_i \), \( \lfloor M \rfloor_{d_i} = \lfloor \lfloor M \rfloor_{d_i} \rfloor_{d_i} \).

Lemma 4 (Selector ordering is irrelevant)
For any two variations \( A \) and \( B \):
\[
\lfloor \lfloor M \rfloor_{B,j} \rfloor_A \equiv \lfloor \lfloor M \rfloor_{A,i} \rfloor_B.
\]

Chen et al. (2014) proved these lemmas for variational types. We can easily adapt those proofs for migrational types by observing that migrational types essentially extend variational types with \( * \)s and \( \lfloor * \rfloor \)s = \( * \). We omit the detailed proof here.

In the proof of Theorem 1, we will use the following lemma.

Lemma 5 (Context filling preserves equivalence)
\[
\lfloor M \rfloor_{\delta} \equiv \lfloor M \rfloor_{\delta} \land \lfloor M[M_1] \rfloor_{\delta} \in V \lor \lfloor M[M_2] \rfloor_{\delta} \in V \Rightarrow \lfloor M[M_1] \rfloor_{\delta} \equiv \lfloor M[M_2] \rfloor_{\delta}.
\]

Proof
By structural induction of the syntax of type contexts \( M[] \).

Case \([ ]\): From the implication in the lemma, we are given the following.
\[
\lfloor M \rfloor_{\delta} \equiv \lfloor M \rfloor_{\delta} \land \lfloor M[M_1] \rfloor_{\delta} \in V \lor \lfloor M[M_2] \rfloor_{\delta} \in V
\]
Since filling the context \([ ]\) with any type yields that type itself, the proof for this case is immediate.

Case \( M'[] \rightarrow M \): We are given the following relations
\[
\lfloor M \rfloor_{\delta} \equiv \lfloor M \rfloor_{\delta} \land \lfloor M[M_1] \rfloor_{\delta} \in V \lor \lfloor M[M_2] \rfloor_{\delta} \in V
\]
Based on induction hypothesis, we have \( \lfloor M'[M_1] \rfloor_{\delta} \equiv \lfloor M'[M_2] \rfloor_{\delta} \). Our goal is to prove the following,
\[
\lfloor M'[M_1] \rightarrow M \rfloor_{\delta} \equiv \lfloor M'[M_2] \rightarrow M \rfloor_{\delta}
\]
which can be transformed to the following based on the definition of selection.
\[
\lfloor M'[M_1] \rfloor_{\delta} \rightarrow \lfloor M \rfloor_{\delta} \equiv \lfloor M'[M_2] \rfloor_{\delta} \rightarrow \lfloor M \rfloor_{\delta}
\]
This equation holds since the domains of both function types are equal based on the induction hypothesis and their codomains are the same. This completes the proof for this case.

Case \( M \rightarrow M'[] \): Similar to the previous case, except that the induction hypothesis and construction deal with the codomain.

Case \( d'M[] \rightarrow M \): We have the following implications and the final equivalence by the induction hypothesis:
\[
\lfloor M \rfloor_{\delta} \equiv \lfloor M \rfloor_{\delta} \land \lfloor M[M_1] \rfloor_{\delta} \in V \lor \lfloor M[M_2] \rfloor_{\delta} \in V \lor \lfloor M'[M_1] \rfloor_{\delta} \equiv \lfloor M'[M_2] \rfloor_{\delta}
\]
We need to show the following,

\[ |d(M'|M_1),M_2|_\delta = |d(M'|M_2),M_1|_\delta \]

We need to consider two subcases. In the first subcase, \(d.1 \in \delta\). Based on Lemmas 3 and 4, the above equation is reduced to the following,

\[ |M'|M_1|_\delta = |M'|M_2|_\delta \]

This follows immediately from the induction hypothesis.

In the second subcase, \(d.2 \in \delta\). Based on Lemmas 3 and 4, the above equation is reduced to the following,

\[ |M|_\delta = |M|_\delta \]

Thus, the lemma holds for this case.

Case \(d(M,M')\): Similar to the previous case and omitted here.

Proof of Theorem 1
Cases MT-Refl-MT-DeadELim are straightforward with the definition of the type equivalence relation in Figure 5. Cases MT-Cong and MT-DynIntro need more care.

Case MT-Cong: We have the following implicants and the final equivalence by the induction hypothesis,

\[ M_1 \approx M_2 \quad M|M_1| \approx M|M_2| \quad |M|M_1| \in V \quad |M|M_2| \in V \quad |M_1|_\delta \equiv |M_2|_\delta \]

and we need to prove the following,

\[ |M|M_1|_\delta \equiv |M|M_2|_\delta \]

The proof is immediate by applying Lemma 5.

Case MT-DynIntro: This case is similar to MT-Cong except that we examine whether \(\delta\) touches the type being inserted into the context. Specifically, if \(|M_1|_\delta\) or \(|M_2|_\delta\) yields a type that contains \(*\), then the implicants of the theorem fail (because implicants require \(|M_1|_\delta\) and \(|M_2|_\delta\) to be variational type that do not contain \(*\)) and the implication holds vacuously. Otherwise, if neither \(|M_1|_\delta\) or \(|M_2|_\delta\) contains \(*\), then \(|M_1|_\delta = |M_2|_\delta\) (because \(M_1\) and \(M_2\) differ by that only \(M_2\) replaced some static types with \(*\)). This case holds due to the reflexivity (VT-refl) of type equivalence.

To prove Theorem 2, we need a lemma similar to Lemma 5 that states that filling type contexts preserves consistency.

Lemma 6 (Context filling preserves consistency)

\[ |M_1|_\delta \sim |M_2|_\delta \wedge |M|M_1|_\delta \in G \wedge |M|M_2|_\delta \in G \Rightarrow |M|M_1|_\delta \sim |M|M_2|_\delta \]

The proof of this lemma is very similar to that of Lemma 5 and is omitted here.

We also need a lemma that captures the type consistency relation among three types. We say a type \(G_2\) is more precise than \(G_1\) if \(G_2\) contains fewer \(*\)'s than \(G_1\) and they agree on the static parts (Garcia & Cimini, 2015). For example, \(\text{Int}\) is more precise than \(*\) and \(\text{Int}\) but not \(\text{Bool}\). As another example, \(\text{Int} \rightarrow \text{Bool}\) is more precise than \(* \rightarrow \text{Bool}\) and \(\text{Int} \rightarrow *\) but not \(* \rightarrow \text{Int}\).
Lemma 7

If $G_1 \sim G_2$, $G_2 \sim G_3$, and $G_2$ is more precise than $G_3$, then $G_1 \sim G_3$.

Proof

By induction on the structures of the involved types.

1. $G_2$ is $\gamma$ or $\alpha$. Based on the definition of $\sim$, rule C1 in Figure 4 applies in this case.

   $G_1$ must be the same as $G_2$, and $G_1 \sim G_3$ holds.

2. $G_2$ is $\ast$. $G_1$ must also be $\ast$, making $G_1 \sim G_3$.

3. $G_2$ has the structure $G_{21} \rightarrow G_{22}$. If either $G_1$ or $G_3$ is $\ast$, then $G_1 \sim G_3$ holds.

   Otherwise, based on rules C1 or C4 of $\sim$, $G_1$ has the structure $G_{11} \rightarrow G_{12}$ and $G_3$ has

   the structure $G_{31} \rightarrow G_{32}$. Moreover, since arrows are covariant on both consistency

   and precision, we have $G_{11} \sim G_{21}$, $G_{21} \sim G_{31}$, and $G_{21}$ more precise than $G_{31}$. We

   thus have $G_{11} \sim G_{31}$. Similarly, we have $G_{12} \sim G_{32}$. Based on rule C4 of $\sim$, we have

   $G_1 \sim G_3$.

We can now prove Theorem 2 that says if two types are compatible then their

Corresponding variants are consistent if they do not contain variations. For example, from

the definition of $\equiv$, we have $A(\text{Int,Bool}) \equiv A(\text{Int,ast})$. Based on that relation, we have

Int $\equiv$ Int at A.1 and Bool $\equiv$ * at A.2.

Proof of Theorem 2

The proof follows by induction over the rules in Figure 8. Cases MT-REFL and MT-SYM

are straightforward via the induction hypotheses and because consistency is reflexive and

symmetric. Case MT-TRANS is also simple since the rule deals with variational types

(without *s) only. As a result, eliminating all variations in types will yield static types,

where the compatibility relation degrades to the equality relation, which is transitive.

Case MT-IDEMP: We are given with the following

$$\left[M\right]_\delta \in G \quad \left[d\left[M,M\right]\right]_\delta \in G$$

and need to show the following implicand.

$$\left[M\right]_\delta \sim \left[d\left[M,M\right]\right]_\delta$$

From $\left[d\left[M,M\right]\right]_\delta \in G$, we know that $d.1 \in \delta$ or $d.2 \in \delta$. Either way, we have

$$\left[d\left[M,M\right]\right]_\delta = \left[M\right]_\delta$$

based on the definition of $\cdot _\delta$. This case thus holds due to rule C1.

Case MT-DEADELIM: We know the following

$$\left[d\left[M_1,M_2\right]\right]_\delta \in G \quad \left[d\left[M_1\right]_{d.1}, \left[M_2\right]_{d.2}\right]_\delta \in G \quad d\left[M_1,M_2\right] \equiv d\left[\left[M_1\right]_\delta, \left[M_2\right]_\delta\right]$$

and we need to prove the following relation.

$$\left[d\left[M_1,M_2\right]\right]_\delta \sim \left[d\left[M_1\right]_{d.1}, \left[M_2\right]_{d.2}\right]_\delta$$

Both $\left[d\left[M_1,M_2\right]\right]_\delta \in G$ and $\left[d\left[M_1\right]_\delta, \left[M_2\right]_\delta\right]_\delta \in G$ imply that either $d.1 \in \delta$ or $d.2 \in \delta$. We assume $d.1 \in \delta$, and we have $\delta = \{d.1\} \cup \delta'$ for some $\delta'$. The proof for when
\[ d.2 \in \delta \] is similar. With Lemma 4, we can move \( d.1 \) to be the first selector used on the types. We then have:

\[
\begin{align*}
|d(M_1, M_2)_\delta| & = |[d(M_1, M_2)]_{d.1}\delta'| \\
& = |[M_1]_{d.1}\delta'| \\
& \quad \text{Definition of } [\cdot]_\delta
\end{align*}
\]

\[
|d(M_1, d.1, M_2)_\delta| = |d(M_1, d.1, [M_2]_{d.2})_{d.1}\delta'| \\
& = |([M_1]_{d.1})_{d.1}\delta'| \\
& \quad \text{Definition of } [\cdot]_\delta
\]

\[ \delta = \{d.1\} \cup \delta' \]

This case thus holds due to rule C1.

Case MT-CONG: This case follows similarly to the case for MT-CONG in the proof for theorem 1.

Case MT-DYNINTRO: We have the following implicands and induction hypothesis,

\[ M_1 \approx M_2[M] \quad [M_1]_\delta \in G \quad [M_2[M]]_\delta \in G \quad [M_1]_\delta \sim [M_2[M]]_\delta \]

and we need to show that

\[ [M_1]_\delta \sim [M_2[\ast]]_\delta \]

First, as \([M_1]_\delta \in G\) and \([M_1]_\delta \sim [M_2[M]]_\delta\), we have \([M_2[M]]_\delta \in G\), implying that \([M]_\delta \in G\). Next, it is obvious that \([\ast]_\delta \in G\) and \([M]_\delta \sim [\ast]_\delta\). Based on Lemma 6, we have \([M_2[M]]_\delta \sim [M_2[\ast]]_\delta\). Moreover, it is obvious that \([M_2[M]]_\delta\) is more precise than \([M_2[\ast]]_\delta\). Based on Lemma 7, we have \([M_1]_\delta \sim [M_2[\ast]]_\delta\).

Before proving Theorem 3, we need two auxiliary lemmas stating that consistent and equivalent types are also compatible. The proof of the first lemma itself makes use of the following lemma.

Lemma 8 (\(\cap\) makes types more precise)

Let \(G_3 = G_1 \cap G_2\), then \(G_3\) is equally or more precise than \(G_1\) and \(G_2\).

The proof of this lemma is a simple induction over the definition of \(\cap\) in Figure 4 and is omitted here.

Lemma 9 (Consistent types are compatible)

\[ G_1 \sim G_2 \Rightarrow G_1 \approx G_2 \]

Proof

The proof proceeds by induction over the definition of consistency in Figure 4.

Case C1: The proof is immediate by applying the rule MT-REFL to the type \(G\).

Case C2: We are given \(G \sim \ast\) and need to derive \(G \approx \ast\). First, we have \(G \approx G\) from the previous case. We can view \(G\) as being obtained by plugging \(G\) into an empty context, thus \(G \approx [G]\). By MT-DYNINTRO, we have \(G \approx [\ast]\), which is the same as \(G \approx \ast\).

Case C3: The proof is the same as the last case followed by applying the rule MT-SYM.

Case C4: We are given:

\[ G_{11} \sim G_{21} \quad G_{12} \sim G_{22} \quad G_{11} \rightarrow G_{12} \sim G_{21} \rightarrow G_{22} \]
and we need to show:
\[ G_{11} \rightarrow G_{12} \approx G_{21} \rightarrow G_{22} \]

First, let \( G_{31} = G_{11} \cap G_{21} \) and \( G_{32} = G_{12} \cap G_{22} \). Through the rule MT-REFL, we have \( G_{31} \rightarrow G_{32} \approx G_{31} \rightarrow G_{32} \). Based on Lemma 8, the type \( G_{31} \) is more static than \( G_{11} \) and \( G_{21} \). Thus, we could repeatedly replace a static component in \( G_{31} \) with a * to reach \( G_{11} \). Based on this observation, we could repeatedly apply the rule MT-DYNINTRO to \( G_{31} \rightarrow G_{32} \approx G_{31} \rightarrow G_{32} \) to get \( G_{31} \rightarrow G_{32} \approx G_{11} \rightarrow G_{12} \). After that, with MT-SYM, we have \( G_{11} \rightarrow G_{12} \approx G_{31} \rightarrow G_{32} \). We can then repeatedly apply MT-DYNINTRO again to prove \( G_{11} \rightarrow G_{12} \approx G_{21} \rightarrow G_{22} \).

\[ \square \]

Lemma 10

\[ V_1 \equiv V_2 \Rightarrow V_1 \approx V_2 \]

Proof

The proof proceeds by induction over the definition of type equivalence in Figure 5. Cases VT-REF, VT-SYM, VT-DEMP, VT-TRANS, and VT-DEADELIM are straightforward, since they are similar in form to MT-REFL-MT-VTTRANS in the definition of compatibility. For this reason, we show the proof for cases VT-CHOICE and VT-FUN only.

Case VT-FUN: We are given:
\[ V_{11} \equiv V_{21} \quad V_{12} \equiv V_{22} \quad V_{11} \rightarrow V_{21} \equiv V_{12} \rightarrow V_{22} \]

and we have the following by the induction hypotheses
\[ V_{11} \approx V_{21} \quad V_{12} \approx V_{22} \]

Next, by using the rule MT-CONG and the first induction hypothesis and setting the context to be \([\square] \rightarrow V_{21}\), we can derive \( V_{11} \rightarrow V_{21} \approx V_{12} \rightarrow V_{21} \). Similarly, by using the rule MT-CONG and the second induction hypothesis and setting the context to be \( V_{12} \rightarrow [\square] \), we can derive \( V_{12} \rightarrow V_{21} \approx V_{12} \rightarrow V_{22} \). Finally, we can use MT-VTTRANS to derive \( V_{11} \rightarrow V_{21} \approx V_{12} \rightarrow V_{22} \).

Case VT-CHOICE: We are given
\[ V_{11} \equiv V_{21} \quad V_{12} \equiv V_{22} \quad d(V_{11}, V_{21}) \equiv d(V_{12}, V_{22}) \]

and have the following by the induction hypotheses:
\[ V_{11} \approx V_{21} \quad V_{12} \approx V_{22} \]

Following the similar proof idea for the last case, we first use the context \( d([\square], V_{21}) \) and the first induction hypothesis to arrive at \( d(V_{11}, V_{21}) \approx d(V_{12}, V_{21}) \). Next, we use the context \( d(V_{12}, [\square]) \) and the second induction hypothesis to derive \( d(V_{12}, V_{21}) \approx d(V_{12}, V_{22}) \). Finally, through MT-VTTRANS we have \( d(V_{11}, V_{21}) \approx d(V_{12}, V_{22}) \).

\[ \square \]

Before proving the theorem, we present a lemma relating types and the types they produce through selection.
*Lemma 11*
\[ \forall \delta. |M_1|_\delta \approx |M_2|_\delta \Rightarrow M_1 \approx M_2 \]

**Proof**
This proof follows by induction on compatibility. Each case is immediate, since applying the induction hypothesis to the premises yields compatible types that can be used to generate the conclusion.

**Proof of Theorem 3**
We directly use Lemmas 9 and 10 to show that all selections producing equivalent or consistent types produce compatible types. We then use Lemma 11 to derive that the types are compatible.

**Proof of Theorem 4:**
The proof follows by induction over the rules in Figure 10.

Case **con**: This case is straightforward because a constant \( c \) always has the plain type \( \gamma \) and \( \forall \delta. |\gamma|_\delta = \gamma \).

Case **var**: The proof is direct from the fact that \( |\Gamma|_\delta \) changes \( x \mapsto M \) in the environment to \( x \mapsto |M|_\delta \). So we can directly use **var** to conclude

\[ \text{statifierForDesc}(\Omega, \delta); [\Gamma]_\delta \vdash_{GC} x : |M|_\delta. \]

Case **abs**: Given the initial typing: \( \pi; \Gamma \vdash \lambda x.e : V \rightarrow M | \Omega \) we want to verify that for any \( \delta \) and some \( \Omega \) where \( |\pi|_\delta = T \), there is a typing:

\[ \text{statifierForDesc}(\Omega, \delta); [\Gamma]_\delta \vdash_{GC} \lambda x.e : [V \rightarrow M]_\delta \]

In the construction of the initial typing, we had the following premise:

\[ \pi; \Gamma, x \mapsto V \vdash e : M | \Omega \]

For this premise, we have the following by the induction hypothesis:

\[ \text{statifierForDesc}(\Omega, \delta); [\Gamma]_\delta, x \mapsto [V]_\delta \vdash_{GC} e : [M]_\delta \]

We then conclude from the result of applying the induction hypothesis and **abs**:

\[ \text{statifierForDesc}(\Omega, \delta); [\Gamma]_\delta \vdash_{GC} \lambda x.e : [V]_\delta \rightarrow [M]_\delta \]

where \( [V]_\delta \rightarrow [M]_\delta = [V \rightarrow M]_\delta \).

Case **absDyn**: Given the initial typing:

\[ \pi; \Gamma \vdash \lambda x : \ast.e : d(\ast, V) \rightarrow M | \Omega \cup \{ x \mapsto V \} \]

we want to show that for any \( \delta \) and some \( \omega \) where \( |\pi|_\delta = T \) we have the following:

\[ \omega; [\Gamma]_\delta \vdash_{GC} \lambda x : \ast.e : \omega(x) \rightarrow [M]_\delta \]

When constructing the initial typing we had the following premise:

\[ \pi; \Gamma, x \mapsto d(\ast, V) \vdash e : M | \Omega \]

For this premise we have the following from the induction hypothesis:

\[ \text{statifierForDesc}(\Omega, \delta); [\Gamma]_\delta, x \mapsto [d(\ast, V)]_\delta \vdash_{GC} e : [M]_\delta \]
Since we do not know whether $\Omega$ has a type for $x$, we must consider whether we can still type $e$ when using $\Omega' = \Omega \cup \{x \mapsto V\}$. We know that $[d \ast, V]_\delta$ must produce either $\ast$ (if $d.1 \in \delta$) or some static type, $T$, where $[V]_\delta = T$. Consequently, we can infer that $\text{statifierForDesc}(\Omega', \delta)(x)$ either produces $\ast$ when $d.1 \in \delta$ or $T$ when $d.2 \in \delta$. Let $\omega = \text{statifierForDesc}(\Omega', \delta)$. We can now derive:

$$\omega; [\Gamma]_\delta, x \mapsto \omega(x) \vdash_G e : [M]_\delta$$

Now we can use $\text{ABSdyn}$ to conclude:

$$\omega; [\Gamma]_\delta \vdash_G \lambda x : \ast. e(x) : [M]_\delta$$

where $\omega(x) = [d \ast, V]_\delta$.

Case $\text{App}$: We are given the initial typing:

$$\pi; \Gamma \vdash e_1 e_2 : \text{cod}_\pi(M_1) \mid \Omega$$

where $\Omega = \Omega_1 \cup \Omega_2$. We need to prove:

$$\text{statifierForDesc}(\Omega, \delta); [\Gamma]_\delta \vdash_G e_1 e_2 : \text{cod}([M_1]_\delta)$$

for any $\delta$ and some $\Omega$ such that $[\pi]_\delta = \top$. In constructing the initial typing we had the following premises:

$$\pi; \Gamma \vdash e_1 : M_1 \mid \Omega_1 \quad \pi; \Gamma \vdash e_2 : M_2 \mid \Omega_2 \quad \text{dom}_\pi(M_1) \approx_\pi M_2$$

We have the following by the induction hypothesis and the premises:

$$\text{statifierForDesc}(\Omega_1, \delta); [\Gamma]_\delta \vdash_G e_1 : [M_1]_\delta \quad \text{statifierForDesc}(\Omega_2, \delta); [\Gamma]_\delta \vdash_G e_2 : [M_2]_\delta$$

Let $\omega = \text{statifierForDesc}(\Omega_1, \delta) \cup \text{statifierForDesc}(\Omega_2, \delta)$, the following two typing relations are satisfied because we can rename parameter names so that $\text{statifierForDesc}(\Omega_1, \delta) (\text{statifierForDesc}(\Omega_2, \delta))$ be a subset of $\omega$ and enlarge $\text{statifierForDesc}(\Omega_1, \delta) (\text{statifierForDesc}(\Omega_2, \delta))$ does not change the typing result of the first (second) typing relation above.

$$\omega; [\Gamma]_\delta \vdash_G e_1 : [M_1]_\delta \quad \omega; [\Gamma]_\delta \vdash_G e_2 : [M_2]_\delta$$

Given that $\pi; \Gamma \vdash e_1 e_2 : \text{cod}_\pi(M_1) \mid \Omega$, we have the following result

$$\text{dom}_\pi(M_1) \approx_\pi M_2 \Rightarrow \text{dom}_\top([M_1]_\delta) \approx_\top [M_2]_\delta \quad [\pi]_\delta = \top \Rightarrow \text{dom}([M_1]_\delta) \sim [M_2]_\delta \quad \text{Theorem 2}$$

We can now use $\text{App}$ to conclude:

$$\omega; [\Gamma]_\delta \vdash_G e_1 e_2 : \text{cod}([M_1]_\delta)$$

Case $\text{If}$: The proof for this case is similar to that for the $\text{App}$ case and is omitted here.

Case $\text{Weaken}$: This rule can only modify selections on $M$ where a decision $\delta'$ yields $[\pi]_\delta = \bot$. Since the theorem requires $[\pi]_\delta = \top$, the proof for this case is vacuous.
A.2 Proofs of Theorems 5 and 6

Proof of Theorem 5:

This proof follows by induction over the rules in Figure 4. The proofs for \textsc{Con}, \textsc{Var}, and \textsc{Abs} are straightforward since they do not introduce variations and have at most one subexpression.

Case \textsc{AbsDyn}: We have the initial typing:

\[
\omega; \Gamma \vdash \omega(x) \rightarrow G
\]

and we need to derive the following:

\[
\pi; \Gamma \vdash \lambda x. e : M' | \Omega
\]

where \( |M'|_\delta = \omega(x) \rightarrow G \) and there is some \( \Omega \) such that \( \text{statifierForDesc}(\Omega, \delta) = \omega \).

We have the following premise when constructing the initial typing:

\[
\omega; \Gamma, x \rightarrow \omega(x) \vdash_{GC} e : G
\]

There are two possibilities for \( x \): either it remains \( \star \) or is updated to a static type.

Since the proof for the first possibility is direct, we focus on the second, where we assume \( x \) is updated to \( T \). By the induction hypothesis and the premise, we have the following:

\[
\pi; \Gamma, x \rightarrow \omega(x) \vdash_{GC} e : M' | \Omega
\]

Based on the first result from applying the induction hypothesis and by the static gradual guarantee, we have:

\[
\pi; \Gamma, x \rightarrow \star \vdash_{d} e : M' | \Omega
\]

Putting the above typing relation and the first induction hypothesis together, we have

\[
\pi; \Gamma, x \rightarrow d(\star, T) \vdash_{d} e : d(M', M) | \Omega \cup \{x \rightarrow T\}
\]

Since the function was well typed in ITGL, we can use \( \top \) when typing the variational version. Then we can use \textsc{AbsDyn} to derive:

\[
\top; \Gamma \vdash \lambda x. e : d(\star, T) \rightarrow d(M', M) | \Omega \cup \{x \rightarrow T\}
\]

Let \( \Omega' = \Omega \cup \{x \rightarrow T\} \). We next show that \( \text{statifierForDesc}(\Omega', \delta) \) and \( \{d(\star, T) \rightarrow d(M', M)\}_\delta = T \rightarrow G \). Since we were considering the case when the parameter was updated to a static type, we have \( d.2 \in \delta \). From the induction hypothesis we have \( \text{statifierForDesc}(\Omega, \delta) = \omega \), and thus \( x \rightarrow V \in \Omega \), where \( \text{statifierForDesc}(\Omega, \delta)(x) = |V|_\delta = T \). Consequently, \( \text{statifierForDesc}(\Omega, \delta) = \text{statifierForDesc}(\Omega', \delta) = \omega \). Moreover, we know that:

\[
|d(\star, T) \rightarrow d(M', M)|_\delta = [d(\star, T)]_\delta \rightarrow [d(M', M)]_\delta
\]

\[
= T \rightarrow |M|_\delta
\]

\[
= T \rightarrow G
\]

I.H

Case \textsc{App}: We have the initial typing:

\[
\omega_1 \cup \omega_2; \Gamma \vdash_{GC} e_1 e_2 : \text{cod}(G_1)
\]
The case for

John Peter Campora III, Sheng Chen, Martin Erwig, and Eric Walkingshaw

Lemma 12

We need to derive:

\[ \pi; \Gamma \vdash e_1 \cdot e_2 : \text{cod}_\pi(M_1) \mid \Omega \]

with \([\text{cod}_\pi(M_1)]_\delta = \text{cod}(G_1)\) and there is some \(\Omega\) such that \(\text{statifierForDesc}(\Omega, \delta) = \omega\). From the derivation of the initial typing we have

the following premises:

\[ \omega_1; \Gamma \vdash_G e_1 : G_1 \quad \omega_2; \Gamma \vdash_G e_2 : G_2 \quad \text{dom}(G_1) \sim G_2 \]

By the induction hypothesis and these premises, we have:

\[ \pi_1; \Gamma \vdash e_1 : M_1 \mid \Omega_1 \]
\[ \pi_2; \Gamma \vdash e_2 : M_2 \mid \Omega_2 \]

Let \(\delta' = \delta_1 \cup \delta_2\), \(\pi' = \pi_1 \cap \pi_2\), and \(\pi\) be a pattern such that \([\pi]_{\delta'} = [\pi']_{\delta'}\) and

\[ \forall \delta, \delta' \neq \delta' \rightarrow [\pi]_{\delta} = \bot. \]

As a result, \([\pi]_{\delta'} = [\pi']_{\delta'} = [\pi_1 \cap \pi_2]_{\delta_1 \cup \delta_2} = [\pi_1]_{\delta_1 \cup \delta_2} \cap [\pi_2]_{\delta_1 \cup \delta_2} = [\top]_{\delta_1 \cup \delta_2} = T \cap T = T. \]

Based on the construction of \(\pi, \pi \leq \pi' \leq \pi_1\). Thus, we have \(\pi; \Gamma \vdash e_1 : M_1 \mid \Omega_1\) based on \(\text{Weaken}\). Similarly, we have \(\pi; \Gamma \vdash e_2 : M_2 \mid \Omega_2\). Moreover, based on \([M_1]_{\delta_1} = G_1\), we have \([M_1]_{\delta} = G_1\) since \(\delta_1 \subseteq \delta\). Similarly, we have \([M_2]_{\delta} = G_2\). From

\[ \text{dom}(G_1) \sim G_2 \]

and the construction of \(\pi\), we have \(\text{dom}_\pi(M_1) \approx \pi M_2\) based on Theorem 3. Therefore, we have \(\pi; \Gamma \vdash e_2 : \text{cod}_\pi(M_1) \mid \Omega\), where \(\Omega = \Omega_1 \cup \Omega_2\).

As \(\Omega_1\) and \(\Omega_2\) are used to type different subexpressions, their domains are disjoint, and so do \(\omega_1\) and \(\omega_2\). As a result \(\text{statifierForDesc}(\Omega, \delta') = \omega\). Since \([M_1]_{\delta_1} = G_1\), we have \([\text{cod}_\pi(M_1)]_\delta = \text{cod}(G_1)\). This completes the proof for this case.

The case for \(\text{If}\) can be proved similarly to the \(\text{App}\) case and is omitted here.

Before we continue to present type system properties, we define an operation \((\sqcup)\) on typing patterns. The operation \(\sqcup\) creates the least upper bound of two patterns of the less-defined partial ordering, defined in Figure 10.

We also state some of its properties and its connection to other relations—which will be used in proofs where more defined typing patterns need to be constructed.

\[ \top \sqcup \pi = \top \]
\[ \bot \sqcup \pi = \pi \]

\[ d(\pi_1, \pi_2) \sqcup d(\pi_3, \pi_4) = d(\pi_1 \sqcup \pi_3, \pi_2 \sqcup \pi_4) \]

\[ d(\pi_1, \pi_2) \sqcup \pi = d(\pi_1 \sqcup \pi, \pi_2 \sqcup \pi) \]

**Lemma 12 (Properties of \((\sqcup)\))**

1. \(\pi_1 \leq \pi \land \pi_2 \leq \pi \Rightarrow \pi_1 \sqcup \pi_2 \leq \pi\)
2. \(M \approx_{\pi_1} M_1 \land M \approx_{\pi_2} M_2 \Rightarrow M \approx_{\pi_1 \sqcup \pi_2} M_1\)
3. \(M \circ_{\pi_1} M_1 \land M \circ_{\pi_2} M_2 \Rightarrow M \circ_{\pi_1 \sqcup \pi_2} M_1\)
4. \(M \circ_{\pi_1}(M) \land M \circ_{\pi_2}(M) \Rightarrow M \circ_{\pi_1 \sqcup \pi_2}(M)\)

The proofs of these properties follow directly from induction over \(\pi\), the definition of \(\sqcup\), the rules for \(\leq\) in Figure 10, and the rules for pattern-constrained operations and compatibility in Figure 9. We omit presenting detailed proofs of these properties for brevity.

**Proof of Lemma 1**
The proof follows from induction over the rules in Figure 10. The cases for Con and Var are straightforward since they can always be typed with the pattern $\top$; the cases for Abs and AbsDyn are also simple because only one subexpression is involved and the proof can be derived simply from the induction hypotheses. We thus omit the proof for these cases.

Case App: We know the following:

$$\pi_1; \Gamma \vdash e_1 \ e_2 : \text{cod}_{\pi_1}(M_1) \mid \Omega \quad \pi_2; \Gamma \vdash e_1 \ e_2 : \text{cod}_{\pi_2}(M_1) \mid \Omega$$

and need to prove the following relation

$$\pi_3; \Gamma \vdash e_1 \ e_2 : \text{cod}_{\pi_3}(M_1) \mid \Omega$$

with $\pi_1 \leq \pi_3$ and $\pi_2 \leq \pi_3$. In the construction of the implicants, we derived the following premises:

$$\pi_1; \Gamma \vdash e_1 : M_1 \mid \Omega \quad \pi_1; \Gamma \vdash e_2 : M_2 \mid \Omega$$
$$\pi_2; \Gamma \vdash e_1 : M_1 \mid \Omega \quad \pi_2; \Gamma \vdash e_2 : M_2 \mid \Omega$$

By the induction hypothesis and these premises, we have:

$$\pi_3; \Gamma \vdash e_1 : M_1 \mid \Omega \quad \pi_3; \Gamma \vdash e_2 : M_2 \mid \Omega$$

$$\pi_3 = \pi_3 \mid \scriptsize{\pi_3'}$$
$$\pi_3 \leq \pi'_3$$

We take $\pi_3 = \pi_1 \sqcup \pi_2$ and we must now show that $\pi_3$ can be used in the typing of the implicant. To type the implicants with $\pi_3$ we know that $\text{dom}_{\pi_1}(M_1)$ and $\text{dom}_{\pi_2}(M_1)$ must be defined. Based on Lemma 12 property 3 and the definition of $\pi_3$, we have:

$$\text{dom}_{\pi_1}(M_1) \land \text{dom}_{\pi_2}(M_1) \Rightarrow \text{dom}_{\pi_3}(M_1)$$

Similarly, we can see that that $\text{dom}_{\pi_1}(M_1) \approx_{\pi_1} M_2$ via property 2. Moreover, $\pi_1 \leq \pi'_3$ and $\pi_2 \leq \pi'_3$ imply that $\pi_3 \leq \pi'_3$, from property 1 in Lemma 12. Consequently, we can use Weaken with $\pi_3$ to derive $\pi_3; \Gamma \vdash e_1 : M_1 \mid \Omega$ and $\pi_3; \Gamma \vdash e_2 : M_2 \mid \Omega$. Pairing those typings with $\text{dom}_{\pi_1}(M_1) \approx_{\pi_1} M_2$, we can use App to conclude:

$$\pi_3; \Gamma \vdash e_1 \ e_2 : \text{cod}_{\pi_3}(M_1) \mid \Omega$$

The proof for the If and Weaken cases follows a similar structure to the App case and is omitted here.

The proof of Lemma 2 relies on Lemmas 13 and 14, which we present first.

Lemma 13

If $M_1 \preceq M_2$ then $\text{cod}_\pi(M_1) \preceq \text{cod}_\pi(M_2)$.

This lemma states that the better relation is preserved when taking the codomain of two types. The proof is straightforward and is omitted here.

The next lemma states that if we can type an abstraction with different static types for the parameter, then we can also type the abstraction with a type that is more general than both of these static types. We first capture the idea of generating a more general static type from two static types with the operation $\sqcap^\alpha$. We define $\sqcap^\alpha$ by extending the definition of $\sqcap$ (Figure 10) with a case $\gamma_1 \sqcap^\alpha \gamma_2 = \alpha$, where $\gamma_1$ and $\gamma_2$ represent two different static constant types. From $\sqcap^\alpha$, we derive $\sqcap^\pi$ as we did for deriving $\sqcap$ from $\sqcap$ and as for deriving $\text{dom}_\pi$ from $\text{dom}$ (Section 4.3).
Lemma 14 (Typing under different assumptions)

For any $e$ and $\Gamma$, if $\pi; \Gamma, x \mapsto d(\ast, V_1) \vdash e : M_1 | \Omega_1$ and $\pi; \Gamma, y \mapsto d(\ast, V_2) \vdash e : M_2 | \Omega_2$, then

$\pi; \Gamma, x \mapsto d(\ast, V_1 \cap_\Omega V_2) \vdash e : M_1 | \Omega_1 \cup \{x \mapsto V_1\}, M_2 \leq M_2, M_1 \leq M_1, \Omega_1 \leq \Omega_1, \text{ and } \Omega_2 \leq \Omega_2$.

In the Lemma, we write $\omega_1 \leq \omega_2$ if $\omega_1$ and $\omega_2$ share the same domain and if for any $x$ in the domain $\omega_1(x) \leq \omega_2(x)$.

The proof is an induction over the typing rules in Figure 10. The case $\text{CON}$ is immediate.

For the case $\text{VAR}$, we need to consider two subcases. The first subcase is that the variable being referenced is not $x$, and the proof is immediate. The second subcase is that the variable being referenced is $x$, then the proof proceeds by observing that $V_1 \leq V_1 \cap_\Omega V_2$ and $V_2 \leq V_1 \cap_\Omega V_2$. The proof for cases $\text{Abs}$ and $\text{AbsDyn}$ are based on simple inductions.

The proof for $\text{APP}$ and $\text{IF}$ is similar to the proof of these cases for Lemma 2 and is omitted here. The proof for $\text{Weaken}$ is based on a simple induction.

Proof of Lemma 2

The proof follows by induction over the rules in Figure 10. Cases $\text{CON}$ and $\text{VAR}$ are straightforward and omitted for brevity. Case $\text{Abs}$ is also omitted since it is similar to $\text{AbsDyn}$, covered below.

Case $\text{AbsDyn}$: We are given with the following:

\[ \pi; \Gamma \vdash \lambda x : \ast. e : d(\ast, V_1) \to M_1 | \Omega_1 \cup \{x \mapsto V_1\} \quad \pi; \Gamma \vdash \lambda x : \ast. e : d(\ast, V_2) \to M_2 | \Omega_2 \cup \{x \mapsto V_2\} \]

We want to show that we can derive the following relation:

\[ \pi; \Gamma \vdash \lambda x : \ast. e : M | \Omega \]

where $d(\ast, V_1) \to M_1 \leq M$ and $d(\ast, V_2) \to M_2 \leq M$ for some $M$ and $\Omega_1 \cup \{x \mapsto V_1\} \leq \Omega$ and $\Omega_2 \cup \{x \mapsto V_2\} \leq \Omega$ for some $\Omega$.

From the construction of the implicants, we know the following premises:

\[ \pi; \Gamma, x \mapsto d(\ast, V_1) \vdash e : M_1 | \Omega_1 \quad \pi; \Gamma, x \mapsto d(\ast, V_2) \vdash e : M_2 | \Omega_2 \]

Based on Lemma 14, let $V_3 = V_1 \cap_\Omega V_2$, we can construct the typing:

\[ \pi; \Gamma, x \mapsto d(\ast, V_3) \vdash e : M_3 | \Omega_3 \]

with $d(\ast, V_1) \leq d(\ast, V_3), d(\ast, V_2) \leq d(\ast, V_3), M_1 \leq M_3, M_2 \leq M_3, \Omega_1 \leq \Omega_3$, and $\Omega_2 \leq \Omega_3$. With $\text{AbsDyn}$, we can derive the following typing relation:

\[ \pi; \Gamma \vdash \lambda x : \ast. e : d(\ast, V_3) \to M_3 | \Omega_3 \cup \{x \mapsto V_3\} \]

Moreover, $d(\ast, V_1) \leq d(\ast, V_3) \to M_3$ and $d(\ast, V_2) \leq d(\ast, V_3) \to M_3$. Let $\Omega_1' = \Omega_1 \cup \{x \mapsto V_1\}, \Omega_2' = \Omega_2 \cup \{x \mapsto V_2\}$, and $\Omega_3' = \Omega_3 \cup \{x \mapsto V_3\}$, we immediately have $\text{statifierForDesc}(\Omega_1', \delta) \leq \text{statifierForDesc}(\Omega_3', \delta) \text{ statifierForDesc}(\Omega_2', \delta) \leq \text{statifierForDesc}(\Omega_3', \delta)$.

Case $\text{APP}$: Given the following judgments:

\[ \pi; \Gamma \vdash e_1 e_2 : \text{cod}_\xi(M_{11}) | \Omega_1 \quad \pi; \Gamma \vdash e_1 e_2 : \text{cod}_\xi(M_{21}) | \Omega_2 \]

we want to prove the following typing derivation:

\[ \pi; \Gamma \vdash e_1 e_2 : \text{cod}_\xi(M_{21}) | \Omega_3 \]
where $\Omega_3$ and $\text{cod}_\pi(M_{31})$ are the best variational statifier and type in the three derivations. In typing the implicants, we had the following premises:

$\pi; \Gamma \vdash e_1 : M_{11} \mid \Omega_{11}$
$\pi; \Gamma \vdash e_2 : M_{12} \mid \Omega_{12}$
$|\pi_{11}|_\delta \preceq |\pi_{12}|_\delta$
$\text{statifierForDesc}(\Omega_{11}, \delta) \preceq \text{statifierForDesc}(\Omega_{31}, \delta)$
$|\pi_{12}|_\delta \preceq |\pi_{12}|_\delta$
$\text{statifierForDesc}(\Omega_{12}, \delta) \preceq \text{statifierForDesc}(\Omega_{32}, \delta)$
$|\pi_{21}|_\delta \preceq |\pi_{21}|_\delta$
$\text{statifierForDesc}(\Omega_{21}, \delta) \preceq \text{statifierForDesc}(\Omega_{32}, \delta)$
$|\pi_{22}|_\delta \preceq |\pi_{22}|_\delta$
$\text{statifierForDesc}(\Omega_{22}, \delta) \preceq \text{statifierForDesc}(\Omega_{32}, \delta)$

Also, note that $\Omega_1 = \Omega_{11} \cup \Omega_{12}$ and $\Omega_2 = \Omega_{21} \cup \Omega_{22}$. We have the following after applying the induction hypothesis:

$\pi; \Gamma \vdash e_1 : M_{31} \mid \Omega_{31}$
$\pi; \Gamma \vdash e_2 : M_{32} \mid \Omega_{32}$
$\text{dom}_\pi(M_{31}) \preceq \text{dom}_\pi(M_{32})$
$\text{statifierForDesc}(\Omega_{11}, \delta) \preceq \text{statifierForDesc}(\Omega_{31}, \delta)$
$\text{statifierForDesc}(\Omega_{12}, \delta) \preceq \text{statifierForDesc}(\Omega_{32}, \delta)$

First we take $\Omega_3 = \Omega_{31} \cup \Omega_{32}$. From our induction hypotheses relating $\Omega_{31}$ and $\Omega_{31}$ to the other statifiers for $e_1$ and $e_3$, it should be clear that we have $\text{statifierForDesc}(\Omega_1, \delta) \preceq \text{statifierForDesc}(\Omega_3, \delta)$ and $\text{statifierForDesc}(\Omega_2, \delta) \preceq \text{statifierForDesc}(\Omega_3, \delta)$.

Now note that $|M_{11}|_\delta \preceq |M_{31}|_\delta$ and $|M_{12}|_\delta \preceq |M_{32}|_\delta$ imply $|\text{cod}_\pi(M_{11})|_\delta \preceq |\text{cod}_\pi(M_{31})|_\delta$ and $|\text{cod}_\pi(M_{12})|_\delta \preceq |\text{cod}_\pi(M_{32})|_\delta$ from Lemma 13. From here, we use our induction hypotheses to derive a return type for the application that is better than the other two.

$\pi; \Gamma \vdash e_1 \ e_2 : \text{cod}_\pi(M_{31}) \mid \Omega_3$

Case II: This case proceeds similarly to APP where most results flow directly from the induction hypotheses.

Case Weaken: Given the following implicant:

$\omega; \Gamma \vdash GC e : M$

We then want to produce the typing derivation:

$\pi_3; \Gamma \vdash e : M_3 \mid \Omega_3$

From deriving the implicant, we know the following from the premises:

$\pi_3; \Gamma \vdash e : M_3 \mid \Omega_3$
$\pi_3 \leq \pi_1 \quad \pi_3 \leq \pi_2$
$M_1 =_{\pi_1} M_1' \quad M_2 =_{\pi_2} M_2'$

By the induction hypothesis and the premises, we have:

$\pi; \Gamma \vdash e : M_3 \mid \Omega_3$
$|M_1|_\delta \preceq |M_3|_\delta \quad \text{statifierForDesc}(\Omega_1, \delta) \preceq \text{statifierForDesc}(\Omega_3, \delta)$
$|M_2|_\delta \preceq |M_3|_\delta \quad \text{statifierForDesc}(\Omega_2, \delta) \preceq \text{statifierForDesc}(\Omega_3, \delta)$

Since we know that $M_3$ is better than the other types, we can always take $\pi_3 = \pi_1 \cap \pi_2$. From there, we can use Weaken to derive:

$\pi_3; \Gamma \vdash e : M_3 \mid \Omega_3$

From here, applying our induction hypothesis to the premises tell us that $\text{statifierForDesc}(\Omega_1, \delta) \preceq \text{statifierForDesc}(\Omega_3, \delta)$ and $\text{statifierForDesc}(\Omega_2, \delta) \preceq \text{statifierForDesc}(\Omega_3, \delta)$.
statifierForDesc(Ω, δ), completing the part of the proof involving statifiers. From here we just need to show \(|M_1'|_δ| ≤ |M_1|_δ| and |M_3'|_δ ≤ |M_3|_δ|

We shall show the first case, and we will omit presenting the second case as it has a similar derivation: We have the following since |π|_δ|=\top:

\[ |M_1|_δ = |M_1'|_δ \]

From this and by our induction hypothesis we can conclude:

\[ |M_1'|_δ ≤ |M_3|_δ \]

Essentially, any selection on \(M_1\) must equal the selection in \(M_1'\) because the \(π\) in both stipulates that the valid selections must produce syntactically equal types. As a result, \(M_3\) is better than the other two types.

\[\square\]

### A.3 Proofs of Theorem 11 and 12

In the following, before presenting each theorem, we present a corresponding lemma that states the property for auxiliary constraint generation functions.

**Lemma 15 (Soundness of Auxiliary Constraint Generation Functions)**

- If \(\text{domCst}(M_a, M_b) \rightarrow C\) and \((θ, π)\) is sound for \(C\), then \(\text{dom}_π(θ(M_a)) \approx_π θ(M_b)\).
- If \(\text{codCst}(M_a) \rightarrow (M_b, C)\) and \((θ, π)\) is sound for \(C\), then \(\text{cod}_π(θ(M_a)) \approx_π θ(M_b)\).
- If \(M_a \cap M_b \rightarrow (M_c, C)\) and \((θ, π)\) is sound for \(C\), then \(θ(M_a) \cap_π θ(M_b) \approx_π θ(M_c)\).

**Proof**

We provide the proof for the first item. The proof for the latter two items is similar and is omitted here. The idea of the proof is going through each case of the function \(\text{domCst}\) and proving the lemma holds.

**Case 1** In this case, we have \(M_a=\star\) and \(M_b=M\). The generated constraint is \(\epsilon\). The sound solution for this constraint is \((θ, \top)\). We know that \(\text{dom}_π(θ(\star))\) is \(\star\), which is compatible with \(θ(M)\).

**Case 2** In this case, \(M_a=α, M_b=M\), and the generated constraint is \(α \approx_? M → κ_2\). Since \((θ, π)\) is sound for this constraint, we have \(θ(α) \approx_π θ(M → κ_2) = θ(M) → θ(κ_2)\). It is immediate that \(\text{dom}_π(θ(M_a)) = θ(M_b)\).

**Case 3** In this case, \(M_a=M_{11} → M_{12}\) and \(M_b=M\). The constraint is \(M_{11} \approx_? M\). By definition, \((θ, π)\) is sound for this constraint means that \(θ(M_{11}) \approx_π θ(M)\). Since \(\text{dom}_π(θ(M_a)) = \text{dom}_π(θ(M_{11} → M_{12})) = θ(M_{11})\), we have \(\text{dom}_π(θ(M_a)) \approx_π θ(M_b)\).

**Case 4** In this case, \(M_a=d(M_1, M_2), M_b=M\), and the constraint is \(d(\text{domCst}(M_1, M), \text{domCst}(M_2, M))\). Assume \((θ, π)\) is a sound solution for the constraint \(d(\text{domCst}(M_1, M), \text{domCst}(M_2, M))\). It will also be sound for \(\text{domCst}(M_1, M)\) and \(\text{domCst}(M_2, M)\). As a result, we have \(\text{dom}_π(θ(M_1)) \approx_π θ(M)\)
Case 5 For any two other types, the constraint is Fail. The sound solution for this constraint is \((\emptyset, \bot)\). Based on the definition of pattern-constrained relations in Figure 9, the relation \(\text{dom}_{\bot}(\theta(M_b)) \approx \bot \theta(M_b)\) holds.

Proof of Theorem 11

The proof proceeds by induction over the constraint generation rules in Figure 11. Cases \(\text{VARC}\) and \(\text{CONC}\) are omitted since they are straightforward.

Case AbsC: Given the following premise:

\[
\Gamma, x \mapsto V \vdash C e : M \mid C
\]

we want to derive:

\[
\pi; \theta(\Gamma) \vdash \lambda x. e : \theta(V \to M) \mid \Omega
\]

We have the following after applying the induction hypothesis to the premise:

\[
\pi; \theta(\Gamma, x \mapsto V) \vdash e : \theta(M) \mid \Omega
\]

where \(\theta(\Gamma, x \mapsto V) = \theta(\Gamma), x \mapsto \theta(V)\). Now applying the Abs typing rule to this judgment, we have

\[
\pi; \theta(\Gamma) \vdash \lambda x. e : \theta(V) \to \theta(M) \mid \Omega
\]

Since \(\theta(V) \to \theta(M) = \theta(V \to M)\), we have

\[
\pi; \theta(\Gamma) \vdash \lambda x. e : \theta(V \to M) \mid \Omega
\]

Case AbsDynC: Proceeds almost identically to Abs.

Case AppC: We are given the judgment \(\Gamma \vdash C e_1 e_2 : M_3 \mid C\), and we have the following premises.

\[
\begin{align*}
\Gamma &\vdash C e_1 : M_1 \mid C_1 \\
\Gamma &\vdash C e_2 : M_2 \mid C_2 \\
\text{domCst}(M_1, M_2) &\to C_4 \\
\text{codCst}(M_1) &\to (M_3, C_3) \\
C &\equiv C_1 \land C_2 \land C_3 \land C_4
\end{align*}
\]

we want to produce the typing derivation:

\[
\pi; \theta(\Gamma) \vdash e_1 : \theta(M_1) \mid \Omega_1
\quad
\pi; \theta(\Gamma) \vdash e_2 : \theta(M_2) \mid \Omega_2
\]

Since \((\theta, \pi)\) is sound for \(C\), it is sound for each \(C_1\) through \(C_4\). Thus, based on the induction hypothesis for the first two premises above, we have

\[
\begin{align*}
\pi; \theta(\Gamma) &\vdash e_1 : \theta(M_1) \mid \Omega_1 \\
\pi; \theta(\Gamma) &\vdash e_2 : \theta(M_2) \mid \Omega_2
\end{align*}
\]
Based on the third premise and Lemma 15 we know that $dom_\pi(M_1) \approx_\pi M_2$. Now, based on the APP rule in Figure 10, we have $\pi, \theta(\Gamma) \vdash e_1 e_2 : \theta(\text{cod}_\pi(M_1)) \mid \Omega$. Based on the fourth premise and Lemma 15 we know that $dom_\pi(M_1) \approx_\pi M_2$, $\theta(\text{cod}_\pi(M_1)) = \theta(M_3)$, which means we have $\pi, \theta(\Gamma) \vdash e_1 e_2 : \theta(M_3) \mid \Omega$.

Case 1C: The proof is similar to APPC and is omitted here.

Now we prove Theorem 12, which investigates the completeness of our constraint generation rules. We arm ourselves with another lemma stating the completeness of the auxiliary constraint generation rules with respect to the definitions of the functions in Figure 10.

**Lemma 16 (Completeness of Auxiliary Constraint Generation)**

- If $dom_\pi(\theta(M_a)) \approx_\pi \theta(M_b)$, $dom\text{Cst}(M_a, M_b) \rightarrow C$, and $(\theta_1, \pi_1)$ is the sound and most general solution for $C$, then $\pi \leq \pi_1$ and $\theta_1 \sqsubseteq \theta$.
- If $\text{cod}_\pi(\theta(M_a)) \approx_\pi \theta(M_b)$, $\text{cod\text{Cst}}(M_a) \rightarrow (M_b, C)$, and $(\theta_1, \pi_1)$ be the sound and most general solution for $C$, then $\pi \leq \pi_1$ and $\theta_1 \sqsubseteq \theta$.
- If $\theta(M_a) \sqcap_\pi \theta(M_b) \approx_\pi \theta(M_c)$, $M_a \sqcap M_b \rightarrow (M_c, C)$, and $(\theta, \pi)$ is sound and most general for $C$, then $\pi \leq \pi_1$ and $\theta_1 \sqsubseteq \theta$.

**Proof**

Again, we prove the first item only. The proof is a case analysis of the definition of $dom$ in Figure 10. Since $dom$ has three cases, so is our proof.

**Case 1** In this case $\theta(M_a) = M_1 \rightarrow M_2$ and $\theta(M_b) = M_1$. We further need to consider two subcases. In the first subcase, $M_a = \alpha$. Based on the definition of $\text{dom\text{Cst}}$, the generated constraint is $\alpha \approx_\pi \theta(M_b)$ as $(\theta_1, \pi_1)$ is sound and most general for this constraint, we have $\theta_1(\alpha) \approx_\pi \theta_1(M_b \rightarrow \kappa_2)$. Since $\kappa_2$ is a fresh unification type variable, $(\theta_1 \sqcup \{\kappa_2 \rightarrow M_2\}, \pi_1)$ is sound and most general for the problem $\alpha \approx_\pi \theta(M_b \rightarrow M_2)$. As $(\theta, \pi)$ is also sound for this problem, $(\theta_1 \sqcup \{\kappa_2 \rightarrow M_2\}, \pi_1)$ is more general than $(\theta, \pi)$. Consequently, $(\theta_1, \pi_1)$ is more general than $(\theta, \pi)$.

In the second subcase, $M_a = M_1' \rightarrow M_2'$. Based on the definition of $\text{dom\text{Cst}}$, the generated constraint is $M_1' \approx_\pi \theta(M_b)$. Since $(\theta_1, \pi_1)$ is most sound and general, we have $\theta_1(M_1') \approx_\pi \theta_1(M_b)$. Moreover, based on the condition of the lemma, we have $dom_\pi(\theta(M_a)) \approx_\pi \theta(M_b)$, meaning that $\theta(M_1') \approx \theta(M_b)$. Overall, both $(\theta_1, \pi_1)$ and $(\theta, \pi)$ are solutions for the same constraint $M_1' \approx_\pi \theta(M_b)$ and $(\theta_1, \pi_1)$ is most general. $(\theta_1, \pi_1)$ is more general than $(\theta, \pi)$.

**Case 2** In this case $\theta(M_a) = \ast$. Since $\theta$ maps type variables to static types only, $M_a = \ast$. Based on the definition of $\text{dom\text{Cst}}$, the generated constraint is $\varepsilon$. The most general solution for it is $(\theta, \top)$, which is more general than $(\theta, \pi)$.

**Case 3** In this case $\theta(M_a) = d(M_1, M_2)$. We again need to consider two subcases, $M_a = \alpha$ and $M_a = d(M_1', M_2')$. The proof for the first subcase is similar to the first subcase of Case 1 above and is omitted here. For the second subcase $dom_\pi(\theta(d(M_1', M_2'))) = dom_\pi(d([\theta]_{d.1}(M_1'), [\theta]_{d.1}(M_2'))) = d(dom_{[\pi]_{d.1}}([\theta]_{d.1}(M_1')), dom_{[\pi]_{d.2}}([\theta]_{d.2}(M_2'))) \approx_\pi \theta(M_b)$. By selecting the both
sides of the compatibility relation with \( d.1 \) and \( d.2 \), we have the following two compatibility results.

\[
\begin{align*}
\text{dom}_{\pi | d.1}([\theta]_{d.1}(M_1')) & \approx_{\pi | d.1} [\theta]_{d.1}(M_b) \quad \text{(A 1)} \\
\text{dom}_{\pi | d.2}([\theta]_{d.2}(M_2')) & \approx_{\pi | d.2} [\theta]_{d.2}(M_b) \quad \text{(A 2)}
\end{align*}
\]

The constraint generated by \( \text{domCst} \) is \( d(M_1', M_2') \approx^2 \), which equals \( d(M_b' \approx^2 M_b, M_b') \) based on the definition of \( \text{domCst} \). Let \((\theta_{1t}, \pi_{1t})\) be the sound and most general solution for \( M_b' \approx^2 \). Based on the equation (1) above and the induction hypothesis, \((\{\theta\}_{d.1}, \{\pi\}_{d.1})\) is more general than \((\{\theta\}_{d.1, \{\pi\}_{d.1}}\)). Similarly, let \((\theta_{1r}, \pi_{1r})\) be the sound and most general solution for \( M_b' \approx^1 \). Based on equation (2) above and the induction hypothesis, \((\theta_{1r}, \pi_{1r})\) is more general than \((\{\theta\}_{d.2}, \{\pi\}_{d.2})\). As a result, \( d(\theta_{1t}, \theta_{1r}, d(\pi_{1t}, \pi_{1r})) \) is sound and most general for \( d(M_1', M_2') \approx^2 \), which is more general than \( d([\theta]_{d.1}, [\theta]_{d.2}, d([\pi]_{d.1}, [\pi]_{d.2})) = (\theta, \pi) \).

\[\square\]

In proving Theorem 12 below, we need to combine several patterns into one. Specifically, given two patterns \( \pi_1 \) and \( \pi_2 \), we calculate their meet \( \pi_1 \sqcap \pi_2 \) as follows (Note, the definition of \( \sqcap \) is also given in Section 7.2, but we reproduced it here for readability).

\[
\begin{align*}
\top \sqcap \pi & = \pi \\
\bot \sqcap \pi & = \bot \\
\pi_1 \sqcap \pi_2 & = \pi
\end{align*}
\]

Intuitively, \( \pi_1 \sqcap \pi_2 \) contains \( \top \)'s at where both \( \pi_1 \) and \( \pi_2 \) contain \( \top \)'s. If either \( \pi_1 \) or \( \pi_2 \) or both contain \( \bot \) at a variant, then \( \pi_1 \sqcap \pi_2 \) also contains a \( \bot \) at that variant. For example, \( \top \sqcap \top = \top \), \( \top \sqcap A(\bot, \top) \) is \( A(\bot, \top) \), and \( A(\bot, \top) \sqcap A(\top, \bot) \) is \( A(\bot, \bot) \), which is the same as \( \bot \).

The operation \( \sqcap \) preserves the less defined relation in the following sense.

\textbf{Lemma 17} \( (\sqcap \text{ preserves the less-defined relation}) \)

If \( \pi \preceq \pi_1 \) and \( \pi \preceq \pi_2 \), then \( \pi \preceq \pi_1 \sqcap \pi_2 \).

The proof is a simple structural induction over the definition of \( \sqcap \) and we omit the detailed proof here.

\textbf{Proof of Theorem 12}

This theorem is proven by structural induction over the rules in Figure 10, with help from Lemma 16. Cases \text{VAR} and \text{CON} are straightforward, so their presentation is omitted.

\textbf{Case ABS: } We are given the following:

\[
\pi; \theta(\Gamma) \vdash \lambda x.e : V \rightarrow M | \Omega, \text{ which has the premise: } \pi; \theta(\Gamma), x \mapsto V \vdash e : M | \Omega
\]

From the premise, we know that there is some type variable \( V' \) such that \( V \preceq V' \) and \( \theta(V') = V \). We thus have \( \pi; \theta(\Gamma, x \mapsto V') \vdash e : M | \Omega \). Based on the induction hypothesis, we have \( \Gamma, x \mapsto V' \vdash C e : M_1 | C \ \forall \delta |_\delta = \top, |M_1|_\delta \preceq |\theta_1(M_1)|_\delta \ \pi \preceq \pi_1 \ \theta = \theta' \circ \theta_1 \).
where $(\theta_1, \pi_1)$ is sound and most general for $C$. With the rule AbsC, we have
\[ \Gamma \vdash_C \lambda x.e : V \to M_1 \mid C, \] where $C$ is the same as that for $e$. Therefore, the solution will be the same and the relation with $(\theta, \pi)$ still hold. Moreover, since $\theta_1$ is more general than $\theta$, $V = \theta(V') \preceq \theta_1(V')$. Therefore, $|V \to M_1|_\delta \preceq |\theta_1(V_1 \to M_1)|_\delta$ based on the induction hypothesis above.

Case AbsDyn: This case proceeds similarly to Abs.

Case App: We are given the following premises:
\[ \pi; \theta(\Gamma) \vdash e_1 : M_1' | \Omega_1, \pi; \theta(\Gamma) \vdash e_2 : M_2' | \Omega_2 \]
\[ \text{dom}_\pi(M_1') \approx_{\pi} M_2' \]
\[ M_3 = \text{cod}_\pi(M_1') \]

We want to derive:
\[ \Gamma \vdash_C e_1 e_2 : M_3 | C \]

such that if $(\theta_1, \pi_1)$ is the solution for $C$, then $\pi \preceq \pi_1$, $\theta_1 \subseteq \theta$, and $M_3' \preceq \theta_1(M_3)$.

We have the following induction hypotheses:
\[ \Gamma \vdash_C e_1 : M_1 | \delta \]
\[ \Gamma \vdash_C e_2 : M_2 | \delta \]
\[ M_1' \preceq \theta_1(M_1) \]
\[ M_2' \preceq \theta_1(M_2) \]
\[ \pi' \preceq \pi_1 \]
\[ \theta = \theta_1' \circ \theta_1 \]

where $(\theta_1, \pi_1)$ solves $C_1$ and $(\theta_1, \pi_12)$ solves $C_2$. From $\text{dom}_\pi(M_1') \approx_{\pi} M_2'$, we have $\text{dom}_\pi(\theta(M_1)) \approx_{\pi} \theta(M_2)$. Let $\text{domCst}(M_1, M_2) \to C_3$ and $(\theta_1, \pi_11)$ be the solution for $C_3$, then, based on Lemma 16, we have $\theta = \theta_1' \circ \theta_13$ and $\pi \preceq \pi_13$. Similarly, from $M_3' \approx_{\pi} \text{cod}_\pi(M_1')$, we have $\theta(M_3) \approx_{\pi} \theta(\text{cod}_\pi(M_1))$. Let $\text{codCst}(M_1) \to C_4$ and $(\theta_14, \pi_14)$ be the solution for $C_4$, then based on Lemma 16, we have $\theta = \theta_1' \circ \theta_14$ and $\pi \preceq \pi_14$ for some $\theta_1'$. We can now use AppC to derive the following relation
\[ \Gamma \vdash_C e_1 e_2 : M_5 | C_1 \land C_2 \land C_3 \land C_4 \]

Moreover, for each $C_i$, we have $(\theta_1, \pi_1)$ that is more general than $(\theta, \pi)$. We next need to prove that we can combine all solutions into one that is still more general than $(\theta, \pi)$. Let $\pi_1 = \pi_11 \cap \pi_12 \cap \pi_13 \cap \pi_14$, we have $\pi \preceq \pi_1$ based on Lemma 17.

We also need to combine $\theta_1$'s. We illustrate the idea by combining $\theta_11$ and $\theta_12$ into $\theta_a$. If $\alpha \to V_a \in \theta_11$ and $\alpha \notin \text{dom}(\theta_12)$, then we add $\alpha \to V_a$ to $\theta_a$. Dually, if $\alpha \to V_b \in \theta_12$ and $\alpha \notin \text{dom}(\theta_11)$, we add $\alpha \to V_b$ to $\theta_a$. If $\alpha \to V_4 \in \theta_11$ and $\alpha \to V_5 \in \theta_12$, then we unify $V_4$ and $V_5$ and add the unified result to $\theta_a$. Since both $\theta_11$ and $\theta_12$ are more general than $\theta$, $\theta_a$ is also more general than $\theta$. Following this idea, let $\theta_1$ be the union of all $\theta_11, \theta_12, \theta_13, \theta_14$, then $\theta_1$ is more general than $\theta$.

Since $\theta_1$ is more general than $\theta$, we have $M_1' \preceq \theta_1(M_1)$. Based on Lemma 13, we have $\text{cod}(M_1') \preceq \text{cod}(\theta_1(M_1))$, which implies that $M_3' \preceq \theta_1(M_3)$.

Case 1f: Similar to App.

A.4 Proofs of Theorems 13 and 14

Proof of Theorem 13

We start by observing that the auxiliary functions $\text{merge}$ and $\text{robinson}$ (for unification) are terminating. The main idea in our proof is that (1) we use a pair $(C_v, C_f)$ to measure the
size of a constraint, where \( C_v \) is the number of unique variations and \( C_f \) is the number of arrows, (2) in each case either \( C_v \) decreases but increases \( C_f \) to a factor of 2, keeps \( C_v \) but decreases \( C_f \), or that case terminates immediately, and (3) when \( (C_v, C_f) \) turns to \((0,0)\), \( \mathcal{U} \) terminates or makes a call to \texttt{robinson}, which is terminating. We go through each case below.

Case (a) This case immediately terminates as no further function calls are made.

Case (a∗) This case is directly delegated to case (a).

Case (b) We consider subcases top-down.

- This subcase immediately terminates as no further function calls are made.
- At first glance, this subcase seems to increase \( C_v \) by 1. However, a close look reveals that this case will be followed by case (c) or (d), which decreases \( C_v \) by 1. This subcase may increase \( C_f \) to a factor of 2.
- The subcase first seems to increase \( C_f \) by 1, but in fact this case will be followed by case (f), which actually decreases \( C_f \) by 1. It does not increase \( C_v \).
- This subcase terminate immediately.

Case (b∗) This case is directly delegated to case (b).

Case (c) This case decrease \( C_v \) by 1 as \( d \) will disappear in the constraint and does not increase \( C_f \).

Case (d) This case decreases \( C_v \) by 1 and increase \( C_f \) by up to a factor of 2, since the type \( M \) appears in one more subproblem.

Case (d∗) This case is directly delegated to case (d).

Case (e) This case will terminate because it calls to \texttt{robinson}, which is terminating.

Case (f) This case decreases \( C_f \) by 1 without increasing \( C_v \).

Case (g) This case immediately terminates.

Case (h) A simple application of induction hypothesis.

Case (i) A simple application of induction hypothesis.

Case (j) This case immediately terminates.

Proof of Theorem 14

By induction on \( \mathcal{U}(M_1 \approx_? M_2) \).

Case (a) and (a∗): Trivial.

Case (b) and (b∗): We consider subcases top-down.

- The substitution is \( \theta = \{ x \mapsto M \} \) with the pattern \( \top \). As \( \theta(\alpha) = M, \theta(\alpha) \approx_\pi \theta(M) \) is clearly satisfied.
- Assume \( (\theta, \pi) = \mathcal{U}(d \langle \alpha, \alpha \rangle) \approx_? M \). By the induction hypothesis, \( \theta(d \langle \alpha, \alpha \rangle) \approx_\pi \theta(M) \). For any \( \delta \) such that \( \theta(d \langle \alpha, \alpha \rangle) \in G \), we have \( \theta(d \langle \alpha, \alpha \rangle) \approx_\pi \theta(M) \). Thus, based on Theorems 1 and 2, Lemmas 9 and 10, and the definition of pattern-constrained relations in Figure 9, we have \( \forall \delta. \theta(\alpha) \approx_\pi \theta(M) \). Now, based on Lemma 11 and pattern-constrained relations, we have \( \theta(\alpha) \approx_\pi \theta(M) \), completing the proof for this subcase.
Case (c): By the induction hypothesis, we have:

\[
\begin{align*}
\theta_1(\alpha) & \simeq_{\pi_1} \theta_1(\kappa_1 \to \kappa_2) \\
\theta_2(\kappa_1 \to \kappa_2) & \simeq_{\pi_2} \theta_2(M_1 \to M_2)
\end{align*}
\]

Moreover, \(\pi_1\) is \(\top\) and \(\theta_1\) does not contain mappings for \(\kappa_1\) and \(\kappa_2\) as they are fresh. Similarly, \(\theta_2\) does not contain a mapping for \(\alpha\) since it does not appear in \(M_1 \to M_2\).

We show that \((\theta_2 \circ \theta_1)(\alpha) \simeq_{\pi_2} (\theta_2 \circ \theta_1)(M_1 \to M_2)\) as follows.

\[
(\theta_2 \circ \theta_1)(\alpha) = \theta_2(\theta_1(\alpha)) = \theta_2(\kappa_1 \to \kappa_2)
\]

\[
= \theta_2(M_1 \to M_2)
\]

\[
\simeq_{\pi_2} \theta_2(\kappa_1 \to \kappa_2)
\]

By (4) above

- The proof is trivial since \(\pi\) is \(\bot\).

Case (c): By the induction hypothesis, we have:

\[
\theta_1(M_1) \simeq_{\pi_1} \theta_1(M_3) \quad \theta_2(M_2) \simeq_{\pi_2} \theta_2(M_4)
\]

Let \(\theta' = \text{merge}(d, \theta_1, \theta_2)\). We need to show: \(\theta'(d|_{M_1, M_2}) \simeq_{d|_{\pi_1, \pi_2}, \theta'}(d|_{M_3, M_4})\).

By Lemma 11, two types are compatible, if any selection on the two types yields compatible relations. We aim to derive the following:

\[
[\theta'(d|_{M_1, M_2})]|_{d.1} \simeq_{[d|_{\pi_1, \pi_2}]|_{d.1}} [\theta'(d|_{M_3, M_4})]|_{d.1}
\]

Because substitution proceeds structurally over choice types, we must show:

\[
[\theta_1(M_1)]|_{d.1} \simeq_{[\pi_1]|_{d.1}} [\theta_1(M_3)]|_{d.1}
\]

To show this, we follow the idea in proving the second subcase of the case (b) above by combining the first induction hypothesis above and Lemma 11. We can similarly prove the case when selecting the target compatibility relation with \(d.2\). As a result, we have

\[
\theta'(d|_{M_1, M_2}) \simeq_{d|_{\pi_1, \pi_2}, \theta'}(d|_{M_3, M_4})
\]

Case (d) and \(d')\): Assume we have \((\theta, \pi) = \varphi(d|_{M_1, M_2}) \approx^\theta d|_{\varphi_1, \varphi_2}(M_1, M_2)\).

By the induction hypothesis, we have: \(\theta(d|_{M_1, M_2}) \approx_{\pi} \theta(d|_{M_1|_{d.1}, M_2|_{d.2}})\).

Our goal is to show:

\[
\theta(d|_{M_1, M_2}) \approx_{\pi} \theta(M)
\]

First, for any \(\delta\) such that \([\theta(d|_{M|_{d.1}, M|_{d.2}})]|_{\delta} \in G\), we have

\[
[\theta(d|_{M|_{d.1}, M|_{d.2}})]|_{\delta} = [\theta(M)]|_{\delta}\]

based on the definition of selection.

Next, based on the induction hypothesis, Theorems 1 and 2, Lemmas 9 and 10, and the definition of pattern-constrained relations in Figure 9, we have

\[
\forall \delta, [\pi]|_{\delta} = \top \Rightarrow [\theta(d|_{M_1, M_2})]|_{\delta} \approx [\theta(d|_{M_1|_{d.1}, M_2|_{d.2}})]|_{\delta} = [\theta(M)]|_{\delta}
\]

Now, based on Lemma 11 and pattern-constrained relations, we have \(\theta(d|_{M_1, M_2}) \approx_{\pi} \theta(M)\), completing the proof for this case.

Cases (e) through (i) are standard and their proof is omitted here.