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 $\star$ (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

ZU064-05-FPR jfp 16 May 2022 9:53

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 $\star$ have dynamic types and those without annotations are inferred to have static types. In the rest of the paper, we say these parameters are *dynamic* and *static*, respectively. This program is adapted from van Keeken (2006) for formatting rows of a table according to a given width by trimming long rows and padding short rows with empty spaces.

```
rowAtI headOrFoot (fixed::⋆) (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 $\star$ annotations. Removing $\star$ annotations turns out to be much trickier than we may expect. First, if we remove all $\star$ 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.

`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 $\star$ 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 $\star$. 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 $\star$ 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 $\star$ 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 $\star$ 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.[2] 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 $\star$ annotations can be removed. In this paper, we mainly consider the *removal* of $\star$ 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/

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

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

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 `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 (`row`) is affected by whether `i` is static or dynamic. The typing of the other two let-bound variables is identical in both cases. Moreover, since the type of `row` is determined to be the same regardless of whether `i` is static or dynamic, the typing of the body of the let-expression is also identical.

This observation suggests that we should reuse typing results while exploring the migration space to determine which ⋆ 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 `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 $\star$ 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 $\star$ annotations from a set of parameters P, then (1) removing $\star$ 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 $\star$ 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 $\star$ annotations does not provide the same ordering. For example, program 3 removes more $\star$ annotations than program 4, but program 4 has a more static type.

The third observation is that, if removing all $\star$ annotations for a set of parameters causes a type error, then removing the $\star$ 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 $\star$ in the current program with a `Int`, `Bool`, or $\star \to \star$. 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 $\lambda x{:}\star.x\ x$. In the first step, three programs are generated, each replacing the $\star$ with a more precise type. The programs $\lambda x{:}\texttt{Int}.x\ x$ and $\lambda x{:}\texttt{Bool}.x\ x$ do not type check. Therefore, they are not migrations of $\lambda x{:}\star.x\ x$. In contrast, the program $\lambda x{:}\star.x\ x$

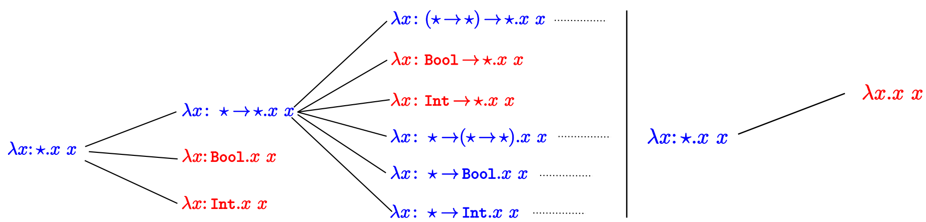

Fig. 2: Programs explored for searching possible migrations in Migeed & Palsberg (2019) (left) and this work (right). Programs in blue type check and those in red do not type check. The dashed lines in the left subfigure denote that an infinite number of programs were omitted from it.

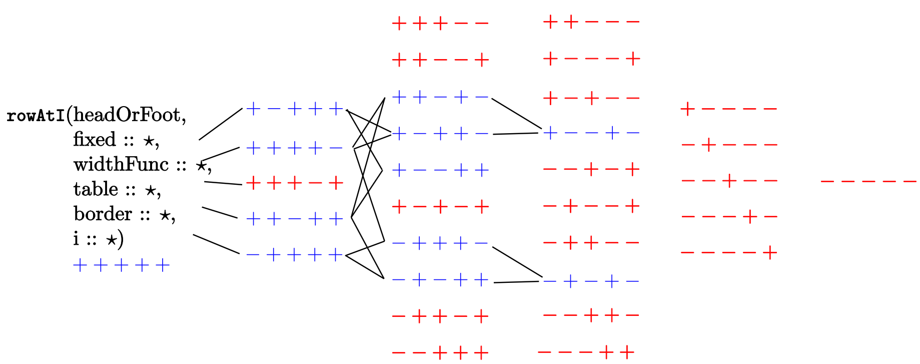

Fig. 3: Programs explored for finding migrations for `rowAtI` in our approach. These programs (configurations) constitute the full migration lattice (Takikawa *et al.*, 2016) for the program `rowAtI`. Each configuration is identified by a sequence of "+/-" signs, with "+" ("-") indicates that the corresponding $\star$ 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 $\star$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 $\star$ 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 $\star$.

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 `rowAtI`. Since `rowAtI` contains five $\star$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 ith character is a $+$, then the ith $\star$ 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 $\star\!\to\!\star$ makes the expressions type checks, and replacing any $\star$ with $\star\!\to\!\star$ 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 $\lambda x\!:\!\star\!\to\!\star.x\ x$. 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: $\beta = \beta_1 \to \beta_2$ and $\beta_1 \sim \beta$, where β, β_1, and β_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 $\star\!\to\!\star$, 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.

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 4.7×10^{-5} seconds to find the max migration for the expression $\lambda x{:}\star.\texttt{succ}(\texttt{succ}\, x)$, 1.5×10^{-4} seconds for the expression $\lambda x{:}\star.\lambda y{:}\star.x+y$, 28.67 seconds for $\lambda x{:}\star.x1:\star.x2:\star.x3:\star.x4:\star.x5:\star.y:\star.y+\texttt{succ}\ (x5\ (x4\ (\texttt{succ}\ x3)(\texttt{succ}\ (x2\ (x1+x+y)))))$,and 93.8 seconds for $\lambda x{:}\star.x1:\star.x2:\star.x3:\star.x4:\star.x5:\star.x6:\star.y:\star.y+\texttt{succ}\ (x5\ (x6+x4\ (\texttt{succ}\ x3)(\texttt{succ}\ (x2\ (x1+x+y)))))$. For these four expressions, our approach takes 4.1×10^{-4}, 5.9×10^{-4}, 1.7×10^{-3}, and 1.9×10^{-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.*,

2012). The type system is correct and efficiently types the whole migration space. We detail the proofs for important cases of the theorems and lemmas that are introduced.
3. In Section 5, we investigate the relationship between different candidate migrations and develop a method for computing the most static migrations.
4. In Sections 6 and 7, we generate and solve constraints to provide type inference for migrational typing and prove that the constraint solving algorithm is correct.
5. In Section 8, we develop a dual to migrational typing to address the migration question Q2.
6. In Section 9, we describe extensions to support additional common language features. We also discuss other migration scenarios and solutions supporting them.
7. In Section 10, we study the performance of our implementation by applying it to synthesized programs. The result shows that our approach scales linearly with program size.

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

Term	Notation	F | P	Operation	Notation	F | P
static types	T	F 7	selection	$\lfloor \cdot \rfloor_{d.1}$	P 13
gradual types	G	F 7	compatibility (M)	$\approx$	F 8
variational types	V	F 7	constrained compatibility (M)	$\approx_\pi$	F 9
migrational types	M	F 7	constrained operation (M)	op_π	F 9
statifier	ω	F 4	better ordering (G)	$\preceq$	P 24
variational statifier	Ω	F 7	more static ordering (G)	$\sqsubseteq$	P 24
choices	$d\langle,\rangle$	P 13	stricter ordering (δ)	$\gg$	P 26
decisions/eliminators	δ	P 13/P 26	less defined ordering (π)	$\leq$	F 10
valid eliminators	δ^v	P 26	pattern meet (π)	$\sqcap$	P 35
typing pattern	$\pi, \top, \bot$	F 9			
unification variables	κ	F 7			

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

Syntax:

$$
\begin{array}{llcl}
\text{Expressions} & e & ::= & c \mid x \mid \lambda x.e \mid \lambda x:\star.e \mid e\ e \mid \textbf{if}\ e\ \textbf{then}\ e\ \textbf{else}\ e \\
\text{Static types} & T & ::= & \gamma \mid \alpha \mid T \to T \\
\text{Gradual types} & G & ::= & \gamma \mid \alpha \mid G \to G \mid \star \\
\text{Statifier} & \omega & ::= & \varnothing \mid \omega, x \mapsto T
\end{array}
$$

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

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

$$
\textsc{AbsDyn}\ \frac{\omega;\Gamma, x \mapsto or(\omega(x), \star) \vdash_{GC} e : G'}{\omega;\Gamma \vdash_{GC} (\lambda x:\star.e) : or(\omega(x), \star) \to G'}
$$

$$
\textsc{App}\ \frac{\omega_1;\Gamma \vdash_{GC} e_1 : G \qquad \omega_2;\Gamma \vdash_{GC} e_2 : G' \qquad dom(G) \sim G'}{\omega_1 \cup \omega_2;\Gamma \vdash_{GC} e_1\ e_2 : cod(G)}
$$

$$
\textsc{If}\ \frac{(\omega_i;\Gamma \vdash_{GC} e_i : G_i)^{i:1..3} \qquad \texttt{Bool} \sim G_1}{\omega_1 \cup \omega_2 \cup \omega_3;\Gamma \vdash_{GC} \textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3 : G_2 \sqcap G_3}
$$

Gradual type consistency:

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

Auxiliary definitions:

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

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

lambda calculus extended with dynamic types. Siek & Vachharajani (2008) and Garcia & Cimini (2015) further investigated gradual typing in the presence of type inference.

In this paper, we consider the migration of programs in implicitly typed gradual languages. In Figure 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, γ over constant types, and α over type variables. There are two cases for abstraction expressions, one where the parameter is annotated by $\star$ 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*, ω, which is used in the full migrational type system later (Section 4). A statifier is a mapping that maps parameter names that have $\star$s 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, ω 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, ABS for unannotated parameters which must have static types, and ABSDYN for annotated parameters which may have dynamic types. In ABSDYN, we use $or(\omega(x),\star)$ to return $\omega(x)$ if $x \in dom(\omega)$ or $\star$ otherwise. Therefore, if ω 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 ω in Figure 4. As such, mappings such as $x \mapsto \star \to \texttt{Int}$ or $y \mapsto \star \to \star$ do not belong to ω. This follows the spirit of Garcia & Cimini (2015) that inferred types should be fully static. Consequently, we can not find an ω to make the expression $\lambda x{:}\star.x\ x$ well typed, even though the expression $\lambda x{:}\star \to \star.x\ x$ 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, `succ`, has static type $\texttt{Int} \to \texttt{Int}$, both of the following programs in our Haskell-like notation should be accepted by gradual typing.

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

The 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 *dom* and *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 $\star \to \star$).

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, $\texttt{Int} \to \texttt{Int} \sim \texttt{Int} \to \star$ but not $\texttt{Int} \to \texttt{Int} \sim \texttt{Bool} \to \star$, 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.

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

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

A *decision* is a set of selectors; we use δ 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, `succ 1` is a plain expression and a variant of the variational expression `vfun`.

A variational expression may contain several choices. Choices with the same name are synchronized and independent otherwise. For example, the variational expression $A\langle\texttt{succ},\texttt{even}\rangle\ A\langle 2,3\rangle$ has two variants, `succ 2` and `even 3`, obtained by the decisions $\{A.1\}$ and $\{A.2\}$, respectively. The program `succ 3` *cannot* be obtained through selection and so is *not* a variant of this expression. On the other hand, the variational expression $A\langle\texttt{succ},\texttt{even}\rangle\ B\langle 2,3\rangle$ has four variants, and we can obtain the variant `succ 3` with the decision $\{A.1, B.2\}$.

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

Variational types are types extended with choices. We define variational types in Figure 5. They include constant types (γ), such as `Int` and `Bool`, type variables (α), 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

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

$$\lfloor \gamma \rfloor_s = \gamma \qquad \lfloor \alpha \rfloor_s = \alpha \qquad \lfloor V_1 \rightarrow V_2 \rfloor_s = \lfloor V_1 \rfloor_s \rightarrow \lfloor V_2 \rfloor_s \qquad \lfloor d\langle V_1, V_2\rangle \rfloor_{d.1} = \lfloor V_1 \rfloor_{d.1}$$

$$\lfloor d\langle V_1, V_2\rangle \rfloor_{d.2} = \lfloor V_2 \rfloor_{d.2} \qquad \lfloor d\langle V_1, V_2\rangle \rfloor_{d_1.i} = d\langle \lfloor V_1 \rfloor_{d_1.i}, \lfloor V_2 \rfloor_{d_1.i}\rangle \qquad \lfloor V \rfloor_{(s:\delta)} = \lfloor \lfloor V \rfloor_s \rfloor_\delta$$

$$\text{VT-REF}\ \frac{}{V \equiv V} \qquad \text{VT-SYM}\ \frac{V_1 \equiv V_2}{V_2 \equiv V_1} \qquad \text{VT-TRANS}\ \frac{V_1 \equiv V_2 \qquad V_2 \equiv V_3}{V_1 \equiv V_3}$$

$$\text{VT-IDEMP}\quad d\langle V, M\rangle \equiv V \qquad \text{VT-DEADELIM}\quad d\langle V_1, V_2\rangle \equiv d\langle \lfloor V_1 \rfloor_{d.1}, \lfloor V_2 \rfloor_{d.2}\rangle$$

$$\text{VT-CHOICE}\ \frac{V_1 \equiv V_1' \qquad V_2 \equiv V_2'}{d\langle V_1, V_2\rangle \equiv d\langle V_1', V_2'\rangle} \qquad \text{VT-FUN}\ \frac{V_1 \equiv V_1' \qquad V_2 \equiv V_2'}{V_1 \rightarrow V_2 \equiv V_1' \rightarrow V_2'}$$

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

selector that has the same choice name ($d.i$) will yield the ith 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, $\lfloor A\langle A\langle \texttt{Int}, \texttt{Bool}\rangle, \texttt{Bool}\rangle \rfloor_{A.1}$ yields $A\langle \texttt{Int}, \texttt{Bool}\rangle$ while `Int` is the expected result, which could be achieved by recursively selecting $A\langle \texttt{Int}, \texttt{Bool}\rangle$ 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 δ.

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

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

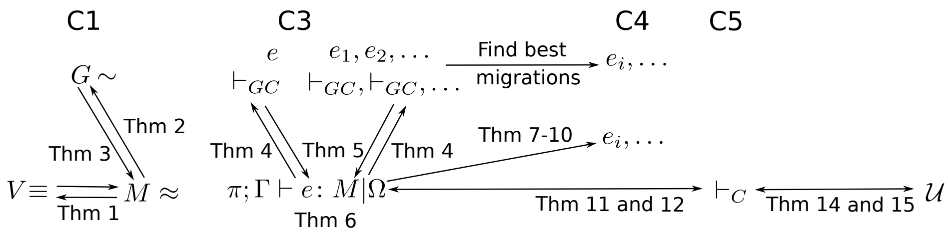

Fig. 6: Relations between theorems and challenges. The notations in the figure are discussed in Section 3.

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

3 Road Map to Migrating Gradual Types

In Section 1.1, we argued that the complexity of the tasks implied by the questions Q1–Q3, involving the migration of gradual programs, is exponential. In Section 2.2, we have shown that variational typing can efficiently type a set of similar programs. A main idea of this paper is to reduce the problem of typing the migration space to variational typing. Specifically, we assign each parameter with a $\star$ annotation a choice type whose the first alternative is a $\star$ and whose second alternative is a static type (In Section 9.1, we deal with parameter types that are partially static, such as $\texttt{Int} \to \star$). Consider, for example, the following function `widthV` that represents the variationally typed version of the function `width` (also shown below) for computing the table width in `rowAtI`.

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

The function `widthV` encodes all four possible migrations of `width`. If $V_{\texttt{widthV}}$ is the type of `widthV`, then $\lfloor V_{\texttt{widthV}} \rfloor_{\{A.1,B.1\}}$ is the type for `width` with no $\star$ annotations removed, $\lfloor V_{\texttt{widthV}} \rfloor_{\{A.2,B.1\}}$ is the type that replaces $\star$ with `Bool` for `fixed` and keeps $\star$ for `widthFunc`, $\lfloor V_{\texttt{widthV}} \rfloor_{\{A.1,B.2\}}$ is the type that keeps $\star$ for `fixed` but replaces $\star$ with $\texttt{Int} \to \texttt{Int}$ for `widthFunc`, and $\lfloor V_{\texttt{widthV}} \rfloor_{\{A.2,B.2\}}$ is the type that removes both $\star$ 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 ($e_1, e_2, \ldots$ in Figure 6) from the given expression (e in the figure) by removing all combinations of $\star$s. 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 `widthV`, we explicitly assigned static types to each parameter. One may wonder whether these are the best types to assign. Maybe other static types could improve the typing result and produce more general types or fewer type errors. 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 $\mathscr{U}$ 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).

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

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

Fig. 7: Syntax of expressions, types, and environments.

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 β to denote the result type of a function application during constraint generation and κ to denote fresh type variables generated during constraint generation and solving (see Sections 6 and 7). For choice names, we typically use A and B to denote arbitrary specific choices in examples and d as a generic metavariable to range over choices names in definitions.

The syntax of expressions, static types, and gradual types are repeated from Section 2.1. To this, we add variational types, which are static types extended with choices, and migrational types, which are gradual types extended with choices. Note that each top-level parameter is assigned a restricted form of migrational type, which is either a fully static type, a $\star$, 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 ω records one way of making a program more static (by removing some subset of $\star$ annotations). A variational statifier Ω instead compactly encodes all possible statifiers for an expression. Since we want migration in our formalization to assign static types to parameters whose $\star$ annotations are removed, Ω maps parameters to variational types, but not migrational types.

Substitutions map type variables to variational types rather than migrational types since substituting dynamic types is unsound. For example, suppose we have `f` $\mapsto \alpha \to \alpha \to \alpha \to \alpha$ and $x \mapsto \star$ in Γ. Now, when typing the application `f x`, we will substitute $\{\alpha \mapsto \star\}$, yielding $\star \to \star \to \star$ as the type of `f x`. However, this implies that `f x 2 True` is well typed, even though this violates the initial static type of `f`. The idea of substituting type variables with variational types but not migrational types is reminiscent

$$\text{MT-REFL}\quad M \approx M \qquad\qquad \text{MT-SYM}\ \frac{M_1 \approx M_2}{M_2 \approx M_1} \qquad\qquad \text{MT-VTTRANS}\ \frac{V_1 \approx V_2 \qquad V_2 \approx V_3}{V_1 \approx V_3}$$

$$\text{MT-IDEMP}\quad d\langle M, M\rangle \approx M \qquad\qquad \text{MT-DEADELIM}\quad d\langle M_1, M_2\rangle \approx d\langle \lfloor M_1 \rfloor_{d.1}, \lfloor M_2 \rfloor_{d.2}\rangle$$

$$\text{MT-CONG}\ \frac{M_1 \approx M_2}{M[M_1] \approx M[M_2]} \qquad\qquad \text{MT-DYNINTRO}\ \frac{M_1 \approx M_2[M]}{M_1 \approx M_2[\star]}$$

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 `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 `fixed` and the application `widthFunc 5` in `widthV`.

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

These two examples demonstrate that we need a notion of *compatibility* between two migrational types to express that all of their variants are consistent. Intuitively, the compatibility relation incorporates both type equivalence for variational types (Chen *et al.*, 2014) and type consistency for gradual types (Siek & 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-VTTRANS) in the case that no $\star$s are present, which we indicate by using the metavariable for variational types (V).

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

The rule 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 $\texttt{Bool} \approx B\langle\texttt{Bool},\texttt{Bool}\rangle$, we have $A\langle\texttt{Int},\texttt{Bool}\rangle \approx A\langle\texttt{Int},B\langle\texttt{Bool},\texttt{Bool}\rangle\rangle$ if we view $A\langle\texttt{Int},[]\rangle$

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

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

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

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

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

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 $\lfloor M \rfloor_\delta \in V$ and $\lfloor M \rfloor_\delta \in G$ to denote that $\lfloor M \rfloor_\delta$ yields a variational type (no $\star$) 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 $\forall \delta. \lfloor M_1 \rfloor_\delta \in V \wedge \lfloor M_2 \rfloor_\delta \in V \Rightarrow \lfloor M_1 \rfloor_\delta \equiv \lfloor M_2 \rfloor_\delta$

Theorem 2 (*Compatibility encodes consistency*)
If $M_1 \approx M_2$, then $\forall \delta. \lfloor M_1 \rfloor_\delta \in G \wedge \lfloor M_2 \rfloor_\delta \in G \Rightarrow \lfloor M_1 \rfloor_\delta \sim \lfloor M_2 \rfloor_\delta$.

Theorem 3 (*Equivalence and consistency imply compatibility*)
$\forall \delta. \lfloor M_1 \rfloor_\delta \equiv \lfloor M_2 \rfloor_\delta \vee \lfloor M_1 \rfloor_\delta \sim \lfloor M_2 \rfloor_\delta \Rightarrow M_1 \approx M_2$

4.3 Pattern-Constrained Judgments

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

While the compatibility check between $A\langle \star, \mathtt{Int} \rangle$ and $B\langle \star, \mathtt{Bool} \rangle$ fails, we observe that $\star$, the first alternative of A, is compatible with $B\langle \star, \mathtt{Bool} \rangle$ and `Int`, the second alternative of A, is compatible with $\star$, the first alternative of B. This suggests that we should

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

$$\lfloor \top \rfloor_\delta = \top \qquad \lfloor \bot \rfloor_\delta = \bot \qquad \lfloor d\langle \pi_1, \pi_2 \rangle \rfloor_{d.1} = \lfloor \pi_1 \rfloor_{d.1} \qquad \lfloor d\langle \pi_1, \pi_2 \rangle \rfloor_{d.2} = \lfloor \pi_2 \rfloor_{d.2}$$

$$\lfloor d\langle \pi_1, \pi_2 \rangle \rfloor_{d_1.i} = d\langle \lfloor \pi_1 \rfloor_{d_1.i}, \lfloor \pi_2 \rfloor_{d_1.i} \rangle \qquad \lfloor \pi \rfloor_{(s:\delta)} = \lfloor \lfloor \pi \rfloor_s \rfloor_\delta$$

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

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

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

describe compatibility at a more fine-grained level than simply saying whether or not two migrational types are compatible. We employ the idea of *typing patterns* (π) (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 $d\langle \pi_1, \pi_2 \rangle$ describes the success or failure of compatibility checking within the context of choice d.

In Figure 9, we also define selection on patterns, which is similar to selection on types ($\lfloor V \rfloor_\delta$) 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 $A\langle \star, \texttt{Int} \rangle$ and $B\langle \star, \texttt{Bool} \rangle$ by the typing pattern $A\langle \top, B\langle \top, \bot \rangle \rangle$. It is also possible to give some pattern that has an identical effect, such as the pattern $B\langle \top, A\langle \top, \bot \rangle \rangle$.

In Figure 9 we define $M_1 \approx_\pi M_2$ such that M_1 and M_2 are compatible for all variants of π that are $\top$. In contrast, there is no requirement between M_1 and M_2 at other places. For example, $\texttt{Int} \approx_{A\langle \bot, \top \rangle} A\langle \texttt{Bool}, \texttt{Int} \rangle$, since $\texttt{Int} \approx \texttt{Int}$ at $A.2$ (and since we do not care that `Int` and `Bool` are incompatible at $A.1$).

The idea of constraining compatibility with patterns is quite powerful. We can even generalize it to typing judgments. Specifically, the typing relation $\pi; \Gamma \vdash e : M$ holds if $\lfloor \Gamma \rfloor_\delta \vdash \lfloor e \rfloor_\delta : \lfloor M \rfloor_\delta$ for all δ such that $\lfloor \pi \rfloor_\delta = \top$. The advantage is that we do not need to worry about the typing in variants where π has $\bot$s. That also means that we should not use (or trust) the typing result at variants where π has $\bot$s. We formally define this relation in Figure 9. For example, since $\Gamma \vdash \texttt{1} : \texttt{Int}$ we have $A\langle \top, \bot \rangle; \Gamma \vdash A\langle \texttt{1}, \texttt{True} \rangle : \texttt{Int}$, even though `True` does not have the type `Int`. We can also generalize this idea to other operations, such as *dom* and *cod*, again defined in Figure 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, *op* could be instantiated to any unary functions, such as *dom* and *cod*. We use the following function *dom* to illustrate this idea.

$$dom(M_1 \to M_2) = M_1 \quad dom(\star) = \star \quad dom(d\langle M_1, M_2\rangle) = d\langle dom(M_1), dom(M_2)\rangle$$

The function *dom* is defined for three cases and is undefined for all other inputs. For example $dom(\texttt{Int} \to \texttt{Bool}) = \texttt{Int}$ but $dom(\texttt{Int})$ is undefined. How about $dom(A\langle \texttt{Int} \to \texttt{Bool}, \texttt{Int}\rangle)$? We can observe that it is defined for the first alternative but not the second alternative. In such case, we can constrain *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 $A\langle \top, \bot\rangle$ 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, $dom_{A\langle \top,\bot\rangle}(A\langle \texttt{Int} \to \texttt{Bool}, \texttt{Int}\rangle)$ is defined in both alternatives of A. Moreover, for the second alternative, we can say the result *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 cod_π if we have a function *cod*, which we define in Figure 10. The rule 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 *dom* is defined above and *cod* will be defined in Figure 10):

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

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

While pattern-constrained judgments simplify the presentation, 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 $A\langle \top, B\langle \top, \bot\rangle\rangle$ in typing `widthFunc fixed` is that the application is ill typed at $\{A.2, B.2\}$. Therefore, in a language with type inference, the pattern will be computed during the inference process (Sections 6 and 7).

4.4 Typing Rules

The typing rules are shown in Figure 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 Γ constrained by the pattern π. The mapping Ω collects the types that will be assigned to parameters

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

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

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

$$\textsc{App}\ \frac{\pi;\Gamma \vdash e_1 : M_1 \,|\, \Omega_1 \qquad \pi;\Gamma \vdash e_2 : M_2 \,|\, \Omega_2 \qquad dom_\pi(M_1) \approx_\pi M_2 \qquad M_3 = cod_\pi(M_1)}{\pi;\Gamma \vdash e_1\ e_2 : M_3 \,|\, \Omega_1 \cup \Omega_2}$$

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

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

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

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

$$\textsc{Pat-Ok}\ \ \pi \leq \top \qquad \textsc{Pat-Err}\ \ \bot \leq \pi \qquad \textsc{Pat-Trans}\ \frac{\pi_1 \leq \pi_2 \qquad \pi_2 \leq \pi_3}{\pi_1 \leq \pi_3} \qquad \textsc{Pat-SinChc}\ \frac{\pi_1 \leq \pi_2 \qquad \pi_1 \leq \pi_3}{\pi_1 \leq d\langle \pi_2, \pi_3\rangle}$$

$$\textsc{Pat-ChcSin}\ \frac{\pi_1 \leq \pi_3 \qquad \pi_2 \leq \pi_3}{d\langle \pi_1, \pi_2\rangle \leq \pi_3} \qquad \textsc{Pat-ChcChc}\ \frac{\pi_1 \leq \pi_3 \qquad \pi_2 \leq \pi_4}{d\langle \pi_1, \pi_2\rangle \leq d\langle \pi_3, \pi_4\rangle}$$

Fig. 10: Typing rules. The operations *dom*, *cod*, and $\sqcap$ are undefined for cases that are not listed here. The process for obtaining dom_π from *dom* is detailed in Section 4.3. The operations cod_π and $\sqcap_\pi$ 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 Ω. The goal of Ω 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 (Con) and variables (Var) are straightforward. They hold for arbitrary patterns π because constants and bound variables are always well typed. Moreover, since the types remain unchanged, Ω is always $\varnothing$. The rule Abs for an abstraction whose parameter is not annotated with $\star$ is conventional. In rule 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 Ω returned from typing the body of the abstraction.

The APP rule for applications is similar to the one in Section 4.3 except that we must combine the variational statifiers from typing the two subexpressions. The operations dom_π and cod_π can be obtained from *dom* and *cod* respectively using the idea of pattern-constrained operations discussed in Section 4.3.

The rule 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 *op* in rule PATBINARY with $\sqcap$. In Section 4.3, we gave a detailed example of deriving dom_π from *dom* and $\sqcap_\pi$ can be derived from $\sqcap$ similarly.

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

The less-defined relation on patterns, written as $\pi_1 \leq \pi_2$, is formally defined in Figure 10. The rules PAT-OK and PAT-ERR define that any pattern is less defined than $\top$ and more defined than $\bot$. The rule PAT-TRANS defines that the relation is transitive. The last three rules handle variational patterns. The rule PAT-SINCHC 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 PAT-CHCSIN states that a variational pattern is less-defined than a pattern if both alternatives are. Finally, the rule PAT-CHCCHC 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 $\star$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 $\star$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 πs and Ms are used for the same Γ and e, which addresses challenge C3 (best typing) from Section 3.

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

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

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

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

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

For example, let

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

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

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

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

We further say that G_2 is *better* than G_1, written as $G_1 \preceq G_2$, if $G_1 \sqsubseteq G_2$ or $G_1 = \theta_2(G_2)$ for some θ_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 $\star \to \alpha \preceq \texttt{Int} \to \texttt{Int}$ and $\texttt{Int} \to \texttt{Int} \preceq \texttt{Int} \to \alpha$.

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

Theorem 4 ($\star$ *removal soundess*)
If $\pi; \Gamma \vdash e : M \,|\, \Omega$, then $\forall \delta. \lfloor \pi \rfloor_\delta = \top \Rightarrow \mathit{statifierForDesc}\,(\Omega, \delta); \lfloor \Gamma \rfloor_\delta \vdash_{GC} e : \lfloor M \rfloor_\delta$.

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 $\pi'_a = A\langle \top, B\langle \top, \bot\rangle\rangle$ we get $\pi'_a; \Gamma \vdash \texttt{width} : M_a \,|\, \Omega_a$, where $M_a = A\langle \star, \texttt{Bool}\rangle \to B\langle \star, \texttt{Int} \to \texttt{Int}\rangle \to B\langle \star, \texttt{Int}\rangle$ and Ω_a is as defined earlier. We can verify $\mathit{statifierForDesc}\,(\Omega_a, \delta_a); \Gamma \vdash_{GC} \texttt{width} : \texttt{Bool} \to \star \to \star$ and $\lfloor M_a \rfloor_{\delta_a} = \texttt{Bool} \to \star \to \star$, where δ_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 $\omega; \Gamma \vdash_{GC} e : G$, then there exists some typing $\pi; \Gamma \vdash e : M \,|\, \Omega$ such that $\lfloor \pi \rfloor_\delta = \top$, $\lfloor M \rfloor_\delta = G$, and $\mathit{statifierForDesc}\,(\Omega, \delta) = \omega$ for some δ.

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 Ω. As a result, the domain of Ωs in different typings are the same. However, the ranges could be different because different typings may use different Vs in AbsDyn.

Lemma 1
If $\pi_1;\Gamma \vdash e : M \,|\, \Omega$ and $\pi_2;\Gamma \vdash e : M \,|\, \Omega$, 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 $\forall \delta. \lfloor \pi \rfloor_\delta = \top \Rightarrow \lfloor M_1 \rfloor_\delta \preceq \lfloor M \rfloor_\delta \wedge \lfloor M_2 \rfloor_\delta \preceq \lfloor M \rfloor_\delta \wedge \mathit{statifierForDesc}(\Omega_1, \delta) \preceq \mathit{statifierForDesc}(\Omega, \delta) \wedge \mathit{statifierForDesc}(\Omega_2, \delta) \preceq \mathit{statifierForDesc}(\Omega, \delta)$.

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 Γ, there is a MGSM typing $\pi;\Gamma \vdash e : M \,|\, \Omega$ such that for any $\pi_1;\Gamma \vdash e : M_1 \,|\, \Omega_1, \forall \delta. \lfloor \pi_1 \rfloor_\delta = \top \Rightarrow \lfloor \pi \rfloor_\delta = \top \wedge \lfloor M_1 \rfloor_\delta \preceq \lfloor M \rfloor_\delta$.

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. □

To illustrate the use of Theorem 6, the MGSM typing for `width` is $\pi_b;\Gamma \vdash \texttt{width} : M_b \,|\, \Omega_b$, where

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

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 δ_2 is *stricter* than an eliminator δ_1, written $\delta_1 \gg \delta_2$, if δ_2 does not select the left alternative (corresponding to $\star$) in more choices than δ_1. Formally,

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

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

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

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

5.1 Relationships Between Migrations

Since every migration can be identified by an eliminator for the MGSM typing, and since stricter eliminators correspond to more static migrations, the problem of 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 π_a from Section 4.5 for `rowAtI` has only 5 eliminators while the expression itself has 32. As another example, from the pattern π_b, defined at the end of Section 4.5 (page 25), we can see that $\delta_{ab}^v = \{A.1\}$ compactly represents δ_a^v and δ_b^v for `width`.

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

Theorem 7 (*Error Irrecoverability*)
Let $\pi; \Gamma \vdash e : M \mid \Omega$ be an MGSM typing for e and Γ. If $\lfloor \pi \rfloor_\delta = \bot$, then $\forall \delta_1 . \delta \gg \delta_1 \Rightarrow \lfloor \pi \rfloor_{\delta_1} = \bot$.

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

Proof

Proof by contradiction. Assume there is some δ_1 such that $\delta \gg \delta_1$ but $\lfloor \pi \rfloor_{\delta_1} = \top$. According to Theorem 4, we have $statifierForDesc(\Omega, \delta_1); \lfloor \Gamma \rfloor_\delta \vdash_{GC} e : \lfloor M \rfloor_{\delta_1}$, which means that e is well typed under the statifier $statifierForDesc(\Omega, \delta_1)$. Based on the definition of statifier generation (Section 4.5), we know that $\delta \gg \delta_1$ implies that $statifierForDesc(\Omega, \delta) \subseteq statifierForDesc(\Omega, \delta_1)$. Therefore, applying $statifierForDesc(\Omega, \delta)$ to e yields a less static expression than $statifierForDesc(\Omega, \delta_1)$ does. Based on the static gradual guarantee for ITGL (Miyazaki *et al.*, 2019), the typing relation $statifierForDesc(\Omega, \delta); \lfloor \Gamma \rfloor_\delta \vdash_{GC} e : \lfloor M \rfloor_\delta$ is satisfied. According to Theorem 6, this implies that $\lfloor \pi \rfloor_\delta = \top$, which contradicts our condition that $\lfloor \pi \rfloor_\delta = \bot$. Therefore, there is no δ_1 such that $\delta \gg \delta_1$ but $\lfloor \pi \rfloor_{\delta_1} = \top$ 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 δ_1 *covers* a valid expression eliminator δ_2 if $\delta_1 \subseteq \delta_2$. Among all the expression eliminators covered by a pattern eliminator, one is the strictest. For example, the eliminator δ^v_{ab} for pattern π_b covers the eliminators δ^v_a and δ^v_b for typing `width`, and δ^v_b is the strictest. As another example, the valid eliminator $\delta^v_{ae} = \{A.1, E.1\}$ for pattern π_a covers eight valid eliminators (two options for each of the three choice names that do not appear in the pattern) for typing `rowAtI`, and $\{A.1, E.1, B.2, D.2, F.2\}$ is the strictest among them.

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

Theorem 8 (*Strict eliminators select better result types*)
If $\pi; \Gamma \vdash e : M \,|\, \Omega$ is the MGSM typing for e and Γ, then $\delta^v_1 \gg \delta^v_2 \wedge \lfloor \pi \rfloor_{\delta^v_1} = \top \wedge \lfloor \pi \rfloor_{\delta^v_2} = \top \Rightarrow \lfloor M \rfloor_{\delta^v_1} \preceq \lfloor M \rfloor_{\delta^v_2}$.

Proof

Based on Theorem 4, we have $statifierForDesc(\Omega, \delta^v_1); \lfloor \Gamma \rfloor_\delta \vdash_{GC} e : \lfloor M \rfloor_{\delta^v_1}$ and $statifierForDesc(\Omega, \delta^v_2); \lfloor \Gamma \rfloor_\delta \vdash_{GC} e : \lfloor M \rfloor_{\delta^v_2}$. Since $\delta^v_1 \gg \delta^v_2$, we have $statifierForDesc(\Omega, \delta^v_1) \subseteq statifierForDesc(\Omega, \delta^v_2)$ based on the definition of statifier generation (Section 4.5). As a result, more precise types are given to variables in a well typed manner and the gradual guarantee (Siek *et al.*, 2015) gives us $\lfloor M \rfloor_{\delta^v_1} \preceq \lfloor M \rfloor_{\delta^v_2}$. □

As an example illustrating Theorem 8, consider δ^v_a, δ^v_b, and M_b, introduced in Section 4.5. We can verify that both $\delta^v_a \gg \delta^v_b$ and $\lfloor M_b \rfloor_{\delta^v_a} \preceq \lfloor M_b \rfloor_{\delta^v_b}$, where $\lfloor M_b \rfloor_{\delta^v_a} = \star \to \star \to \star$, and $\lfloor M_b \rfloor_{\delta^v_b} = \texttt{Bool} \to \star \to \star$.

Theorem 8 provides a way to order the eliminators covered by a single pattern eliminator, but how about ordering different valid eliminators of the typing pattern? Considering pattern π_b, neither of the valid eliminators δ^v_b or δ^v_c is stricter than the other. Similarly, for pattern π_a, neither of the valid eliminators is stricter than the other. In fact, this property holds not only for these two examples, but also for a class of typing patterns that are in *pattern normal form*. We say a pattern is in normal form if it does not contain idempotent choices (choices with identical alternatives) and does not nest a choice in another choice with the same name (no dead alternatives). We capture this property in the following theorem.

Theorem 9 (*Eliminator Incomparability*)

ZU064-05-FPR jfp 16 May 2022 9:53

Let $\pi;\Gamma \vdash e : M \,|\, \Omega$ be MGSM typing for e and Γ and π is in normal form, then $\nexists \delta^v . \delta_1^v \gg \delta^v \wedge \delta_2^v \gg \delta^v$ if δ_1^v and δ_2^v are distinct.

Proof of Theorem 9
Proof by contradiction. Assume there exists such a δ^v. First, δ_1^v contains at least one selector of the form $d.1$ for some d. Otherwise, the program can be fully migrated to be static, and the typing pattern will be $\top$, making δ_1^v and δ_2^v be the same. Similarly, this holds for δ_2^v. Without loss of generality, we assume δ_1^v contains $d_1.1$ and δ_2^v contains $d_2.1$. 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 δ_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 δ_1^v and δ_3^v, we can infer that both alternatives of d_2 are $\top$, meaning that it is an idempotent variation and π 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 $\delta_1^v \gg \delta \wedge \delta_2^v \gg \delta$. If δ is a valid eliminator, then we can again show that d_2 is idempotent, a contradiction that π is in normal form.

We could swap the assignments to δ_1^v and δ_2^v, but this will yield the same proof result. □

It follows from the theorem that for any two valid eliminators δ_1^v and δ_2^v for π_1, $\delta_1^v \not\gg \delta_2^v$ and $\delta_2^v \not\gg \delta_1^v$. Two eliminators that are incomparable with respect to $\gg$ will remove $\star$s for different parameters for the same expression, leading to types that are incomparable by $\sqsubseteq$ (defined in Section 4), and thus incomparable by $\preceq$. For example, since $\delta_b^v \not\gg \delta_c^v$ and $\delta_c^v \not\gg \delta_b^v$, we have $G_b \not\preceq G_c$ and $G_c \not\preceq G_b$, where $G_b = \lfloor M_b \rfloor_{\delta_b^v} = \star \rightarrow (\texttt{Int} \rightarrow \beta) \rightarrow \beta$ and $G_c = \lfloor M_b \rfloor_{\delta_c^v} = \texttt{Bool} \rightarrow \star \rightarrow \star$.

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 $\pi;\Gamma \vdash e : M \,|\, \Omega$ be the MGSM typing for e and Γ, and π is in normal form. Then the number of most static migrations for e equals the number of valid eliminators for π.

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 π_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 $\star$ annotation or the inferred type for each parameter. We compute this set of eliminators from an MGSM typing in three steps: 1. simplify the typing pattern to its normal form, 2. collect the valid eliminators for the normal form, and 3. expand each valid eliminator into a strictest eliminator for the corresponding expression.

Simplifying a typing pattern to its normal form has two advantages. First, the valid eliminators are fewer and smaller. Second, we can use the result of Theorem 10 to find most static migrations. We use the following rules to simplify patterns to normal forms.

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

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

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

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

To illustrate the definition of ve, we consider the calculation process for the pattern $A\langle \top, \bot \rangle$. $ve(A\langle \top, \bot \rangle) = \{\{A.1\} \cup l \mid l \in ve(\top)\} \cup \{\{A.2\} \cup r \mid r \in ve(\bot)\} = \{\{A.1\} \cup l \mid l \in \{\varnothing\}\} \cup \{\{A.2\} \cup r \mid r \in \varnothing\} = \{\{A.1\}\} \cup \varnothing = \{\{A.1\}\}$. This means that the set of valid eliminators for $A\langle \top, \bot \rangle$ contains only one element: $\{A.1\}$. Similarly, $ve(A\langle \bot, \top \rangle) = \{\{A.2\}\}$. As another example, $ve(\pi_a)$ yields $\{\delta^v_o, \delta^v_p\}$, where $\delta^v_o = \{A.1, E.1\}$ and $\delta^v_p = \{A.2, B.1, E.1\}$.

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

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

For example, the set of choice names $\mathscr{D}$ for typing `rowAtI` is $\{A, B, D, E, F\}$, and $expand(\delta^v_o, \mathscr{D})$ yields $\{A.1, E.1, B.2, D.2, F.2\}$ and $expand(\delta^v_p, \mathscr{D})$ 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 $\star$ annotation for `widthFunc`, `table`, and `i`, while the other best eliminator removes the $\star$ annotation for `fixed`, `table`, and `i`.

Formally, given an expression e and its MGSM typing $\pi; \Gamma \vdash e : M \mid \Omega$, then for any expanded valid eliminator δ^v, we can generate the most static migration using $statifierForDesc(\Omega, \delta^v)$, 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 $\star$ 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 $\star$ annotation is removed, then there is again just one valid eliminator.

$$\boxed{\Gamma \vdash_C e : M \mid C}$$

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

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

$$\textsc{AppC}\ \frac{\begin{array}{c}\Gamma \vdash_C e_1 : M_1 \mid C_1 \qquad \Gamma \vdash_C e_2 : M_2 \mid C_2 \\ \mathit{codCst}(M_1) \hookrightarrow (M_3, C_3) \qquad \mathit{domCst}(M_1, M_2) \hookrightarrow C_4 \qquad C = C_1 \wedge C_2 \wedge C_3 \wedge C_4\end{array}}{\Gamma \vdash_C e_1\ e_2 : M_3 \mid C}$$

$$\textsc{IfC}\ \frac{\begin{array}{c}\Gamma \vdash_C e_1 : M_1 \mid C_1 \qquad \Gamma \vdash_C e_2 : M_2 \mid C_2 \\ \Gamma \vdash_C e_3 : M_3 \mid C_3 \qquad M_2 \sqcap M_3 \hookrightarrow (M_4, C_4) \qquad C = C_1 \wedge C_2 \wedge C_3 \wedge C_4 \wedge M_1 \approx^? \texttt{Bool}\end{array}}{\Gamma \vdash_C \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 : M_4 \mid C}$$

Fig. 11: Constraint generation rules

$$\begin{array}{rlrl}
\mathit{domCst}(\star, M) & \hookrightarrow \varepsilon & \mathit{domCst}(\alpha, M) & \hookrightarrow \alpha \approx^? M \to \kappa_2 \\
\mathit{domCst}(M_{11} \to M_{12}, M) & \hookrightarrow M_{11} \approx^? M & \mathit{domCst}(d\langle M_1, M_2 \rangle, M) & \hookrightarrow d\langle \mathit{domCst}(M_1, M), \mathit{domCst}(M_2, M) \rangle \\
\mathit{domCst}(_, _) & \hookrightarrow \texttt{Fail} & & \\
\mathit{codCst}(\star) & \hookrightarrow (\star, \varepsilon) & \mathit{codCst}(\alpha) & \hookrightarrow (\kappa_2, \alpha \approx^? \kappa_1 \to \kappa_2) \\
\mathit{codCst}(M_1 \to M_2) & \hookrightarrow (M_2, \varepsilon) & \mathit{codCst}(d\langle M_1, M_2 \rangle) & \hookrightarrow d\langle \mathit{codCst}(M_1), \mathit{codCst}(M_2) \rangle \\
\mathit{codCst}(_) & \hookrightarrow (\kappa, \texttt{Fail}) & & \\
\alpha \sqcap M & \hookrightarrow (\alpha, \alpha \approx^? M) & d\langle M_1, M_2 \rangle \sqcap M & \hookrightarrow d\langle M_1 \sqcap M, M_2 \sqcap M \rangle \\
M \sqcap \alpha & \hookrightarrow (\alpha, \alpha \approx^? M) & M \sqcap d\langle M_1, M_2 \rangle & \hookrightarrow d\langle M_1, M_2 \rangle \sqcap M \\
\star \sqcap M & \hookrightarrow (M, \varepsilon) & M_{11} \to M_{12} \sqcap M_{21} \to M_{22} & \hookrightarrow (M_1 \to M_2, C_1 \wedge C_2) \\
M \sqcap \star & \hookrightarrow (M, \varepsilon) & & \text{where } M_{11} \sqcap M_{21} \hookrightarrow (M_1, C_1) \\
_ \sqcap _ & \hookrightarrow (\kappa, \texttt{Fail}) & & M_{12} \sqcap M_{22} \hookrightarrow (M_2, C_2)
\end{array}$$

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 $\Gamma \vdash_C e : M \mid C$ states that under Γ, the expression e has type M when the constraint C is solved. Accordingly, e and Γ are inputs, while M and C are outputs. Note that we now omit the statifier Ω in constraint judgments since it is not needed for type inference. We also omit π since π 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:

$$C ::= M_1 \approx^? M_2 \ \mid\ C \wedge C \ \mid\ d\langle C, C \rangle \ \mid\ \varepsilon \ \mid\ \texttt{Fail}$$

The first form represents type compatibility constraints. Often it is the case that two types are only partially compatible. Note, when $M_1 \approx^? M_2$ 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 $C_1 \wedge C_2$ defines the conjunction of two constraints C_1 and C_2, while the constraint $d\langle C_1, C_2 \rangle$ defines a choice between two constraints. The constraint ε represents an empty constraint. This is needed to represent a judgment where no constraints are generated.

Finally, the constraint `Fail` represents a constraint that, when solved, always leads to a failure. Such a constraint is needed when, for example, $dom(\texttt{Int})$ is calculated during the constraint generation process. As `Int` is not a function type, $dom(\texttt{Int})$ will always fail. We generate a `Fail` to communicate this failure to the constraint solver. The constraint `Fail` was absent from the original paper (Campora *et al.*, 2018a). Without it, that work outputs a typing pattern and returns a $\perp$ 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 `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 `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 ConC, generating constraints for constants, has a very similar form to Con in Figure 10. The rule VarC for variable references is similar to Var and, like ConC, generates the empty constraint.

The rule AbsDynC generates constraints for abstractions with dynamic parameters. It helps facilitate migration by creating a fresh choice type with a left alternative containing $\star$ 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 AppC and IfC are more involved because constraints from premises have to be combined. The rules AppC and IfC use many auxiliary functions to generate constraints. The functions, defined in Figure 12, take the form: $domCst(M_1, M_2) \hookrightarrow C$, $codCst(M_1) \hookrightarrow (M_2, C)$, and $M_1 \sqcap M_2 \hookrightarrow (M_3, C)$, where the objects to the left of $\hookrightarrow$ are inputs and those to the right are outputs. Essentially, they implement the *dom*, *cod*, and $\sqcap$ operations defined for the declarative type system in Figure 10. Note, in these functions κ denote fresh type variables. We will use such variables in this and next sections.

We illustrate *domCst* by considering the example $domCst(A\langle \star, \alpha \rangle, \texttt{Int})$. Since the first argument is a choice type, *domCst* proceeds to recursively call on each alternative of A, leading to two subproblems $domCst(\star, \texttt{Int})$ and $domCst(\alpha, \texttt{Int})$. The first subproblem is handled by the case for $\star$, which immediately returns ε, meaning that no further constraints need to be solved. The second subproblem is handled by the case of *domCst* for type variables. Since *dom* always expects a function type, the constraint $\alpha \approx^? \texttt{Int} \rightarrow \kappa_2$ is generated. The constraints for subproblems are combined together with the choice A, yielding the final constraint $A\langle \varepsilon, \alpha \approx^? \texttt{Int} \rightarrow \kappa_2 \rangle$.

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 typing. Following the spirit of Vytiniotis *et al.* (2011), we use the idea of sound and most-general solutions (θ) for constraints (C) in the following theorems (Vytiniotis *et al.* (2011) used the term *guess-free*). (θ,π) is sound for a constraint of the form $M_1 \approx^? M_2$ if $\theta(M_1) \approx_\pi \theta(M_2)$, is sound for a constraint $C_1 \wedge C_2$ or $d\langle C_1, C_2\rangle$ if it is sound for both C_1 and C_2, is sound for `Fail` if π is $\bot$, and is always sound for ε. In Section 7, we provide a unification algorithm that generates solutions with these desired properties.

Theorem 11 (*Soundness of Constraint Generation*)
If $\Gamma \vdash_C e : M \mid C$, then $\pi;\theta(\Gamma) \vdash e : \theta(M) \,|\, \Omega$ for some Ω, where (θ,π) is a sound solution for C.

Theorem 12 (*Completeness of Constraint Generation*)
If $\pi;\theta(\Gamma) \vdash e : M \,|\, \Omega$ then $\Gamma \vdash_C e : M_1 \mid C$ such that $\pi \leq \pi_1$, $\forall \delta. \lfloor \pi \rfloor_\delta = \top \Rightarrow \lfloor \pi_1 \rfloor_\delta = \top \wedge \lfloor M \rfloor_\delta \preceq \lfloor \theta_1(M_1) \rfloor_\delta \wedge \lfloor \theta \rfloor_\delta = \lfloor \theta' \rfloor_\delta \circ \lfloor \theta_1 \rfloor_\delta$ for some θ', where (θ_1,π_1) is a sound and most-general solution for C.

In the theorem, we define $\lfloor \theta \rfloor_\delta$ as $\{\alpha \mapsto \lfloor V \rfloor_\delta \mid \alpha \mapsto V \in \theta\}$.

Two constraint generation examples The following table lists the constraint generation process for the expression $\lambda x{:}\star.$`succ` $(x$ `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 α, respectively.

Subexpression	M (Type)	C (Constraint)
x	$A\langle\star,\alpha\rangle$	ε
`True`	`Bool`	ε
x `True`	$A\langle\star,\kappa_2\rangle$	$A\langle\varepsilon, C_1 \wedge C_2\rangle$
`succ`	`Int` $\to$ `Int`	ε
`succ` $(x$ `True`$)$	`Int`	$A\langle\varepsilon, C_1 \wedge C_2 \wedge C_4\rangle$
$\lambda x{:}\star.$`succ` $(x$ `True`$)$	$A\langle\star,\alpha\rangle \to$ `Int`	$A\langle\varepsilon, C_1 \wedge C_2 \wedge C_4\rangle$

$$C_1 = \alpha \approx^? \kappa_1 \to \kappa_2 \quad C_2 = \alpha \approx^? \texttt{Bool} \to \kappa_4 \quad C_4 = \texttt{Int} \approx^? \kappa_2$$

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 tale lists additional constraints that will be generated from the expression $\lambda x{:}\star.x($`succ` $(x$ `True`$))$.

Subexpression	M (Type)	C (Constraint)
x	$A\langle\star,\alpha\rangle$	ε
x $($`succ` $(x$ `True`$))$	$A\langle\star,\kappa_6\rangle$	$A\langle\varepsilon, C_1 \wedge C_2 \wedge C_4 \wedge C_5 \wedge C_6\rangle$
$\lambda x{:}\star.x$ $($`succ` $(x$ `True`$))$	$A\langle\star,\alpha\rangle \to A\langle\star,\kappa_6\rangle$	$A\langle\varepsilon, C_1 \wedge C_2 \wedge C_4 \wedge C_5 \wedge C_6\rangle$

$$C_5 = \alpha \approx^? \kappa_5 \to \kappa_6 \quad C_6 = \alpha \approx^? \texttt{Int} \to \kappa_8$$

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) $\alpha \approx^? \star \rightarrow \texttt{Int}$
(ii) $A\langle\star,\texttt{Bool}\rangle \approx^? \texttt{Int}$

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, α should be substituted by a function type whose codomain is `Int`, but what should the domain be? Essentially, the domain should be an unconstrained type variable so that it can unify with a static type later, if necessary. As a result, we generate the substitutions $\{\kappa_2 \mapsto \texttt{Int}\} \circ \{\alpha \mapsto \kappa_1 \rightarrow \kappa_2\}$. Since κ_1 is a fresh type variable that is not mapped to anything, it is unconstrained. In contrast, κ_2 is mapped to `Int`. This substitution satisfies both rules (I) and (II).

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

7.2 A Unification Algorithm

Figure 13 presents a unification algorithm $\mathscr{U}$, which takes a constraint and produces a substitution θ and a pattern π. 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 $\perp$. 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 $\star$s and variations.

In the figure, we use the following conventions and helper functions. We use κs to denote fresh type variables. The function *choices*(M) returns the set of choice names in M; *vars*(M) returns the set of type variables in V. The predicate *hasDyn*(M) determines

$\mathscr{U} : C \to \theta \times \pi$

(a) $\mathscr{U}(\star \approx^? M) = (\emptyset, \top)$

(a*) $\mathscr{U}(M \approx^? \star) = \mathscr{U}(\star \approx^? M)$

(b) $\mathscr{U}(\alpha \approx^? M)$

$\quad \mid \alpha \notin vars(M) \wedge \neg hasDyn(M) = (\{\alpha \mapsto M\}, \top)$

$\quad \mid d \in choices(M) = \mathscr{U}(d\langle \alpha, \alpha \rangle \approx^? M)$

$\quad \mid \alpha \notin vars(M) \wedge M$ is of form $M_1 \to M_2 =$

$\qquad$ let $(\theta_1, \pi_1) = \mathscr{U}(\alpha \approx^? \kappa_1 \to \kappa_2)$; $(\theta_2, \pi_2) = \mathscr{U}(\kappa_1 \to \kappa_2 \approx^? M_1 \to M_2)$ in $(\theta_2 \circ \theta_1, \pi_2 \sqcap \pi_1)$

$\quad \mid$ otherwise $= (\emptyset, \bot)$

(b*) $\mathscr{U}(M \approx^? \alpha) = \mathscr{U}(\alpha \approx^? M)$

(c) $\mathscr{U}(d\langle M_1, M_2 \rangle \approx^? d\langle M_3, M_4 \rangle) =$

$\quad$ let $(\theta_1, \pi_1) = \mathscr{U}(M_1 \approx^? M_3)$; $(\theta_2, \pi_2) = \mathscr{U}(M_2 \approx^? M_4)$; $\theta' = merge(d, \theta_1, \theta_2)$

$\quad$ in $(\theta', d\langle \pi_1, \pi_2 \rangle)$

(d) $\mathscr{U}(d\langle M_1, M_2 \rangle \approx^? M) =$

$\quad$ let $(\theta_1, \pi_1) = \mathscr{U}(M_1 \approx^? \lfloor M \rfloor_{d.1})$; $(\theta_2, \pi_2) = \mathscr{U}(M_2 \approx^? \lfloor M \rfloor_{d.2})$; $\theta' = merge(d, \theta_1, \theta_2)$

$\quad$ in $(\theta', d\langle \pi_1, \pi_2 \rangle)$

(d*) $\mathscr{U}(M \approx^? d\langle M_1, M_2 \rangle) = \mathscr{U}(d\langle M_1, M_2 \rangle \approx^? M)$

(e) $\mathscr{U}(T_1 \approx^? T_2) =$ if $robinson(T_1, T_2) = \theta'$ then $(\theta', \top)$ else $(\emptyset, \bot)$

(f) $\mathscr{U}(M_{11} \to M_{12} \approx^? M_{21} \to M_{22}) =$

$\quad$ let $(\theta_1, \pi_1) = \mathscr{U}(M_{11} \approx^? M_{21})$; $(\theta_2, \pi_2) = \mathscr{U}(\theta_1(M_{12}) \approx^? \theta_1(M_{22}))$ in $(\theta_2 \circ \theta_1, \pi_1 \sqcap \pi_2)$

(g) $\mathscr{U}(\varepsilon) = (\emptyset, \top)$

(h) $\mathscr{U}(d\langle C_1, C_2 \rangle) =$

$\quad$ let $(\theta_1, \pi_1) = \mathscr{U}(C_1)$; $(\theta_2, \pi_2) = \mathscr{U}(C_2)$; $\theta' = merge(d, \theta_1, \theta_2)$

$\quad$ in $(\theta', d\langle \pi_1, \pi_2 \rangle)$

(i) $\mathscr{U}(C_1 \wedge C_2) =$ let $(\theta_1, \pi_1) = \mathscr{U}(C_1)$; $(\theta_2, \pi_2) = \mathscr{U}(\theta_1(C_2))$ in $(\theta_2 \circ \theta_1, \pi_2 \sqcap \pi_1)$

(j) $\mathscr{U}(\texttt{Fail}) = (\varnothing, \bot)$

Fig. 13: A unification algorithm.

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

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

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

Intuitively, if $\alpha \in dom(\theta)$, then $get(\alpha, \theta)$ returns the image of α in θ. Otherwise, $get(\alpha, \theta)$ returns a fresh type variable. Recall that κ denotes a fresh type variable.

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

Case (b) unifies a type variable α with a migrational type M. This case includes many subcases. First, if M does not contain $\star$ and α does not occur in M, then α is directly mapped to M. For example, given $\alpha \approx^? A\langle \texttt{Int}, \texttt{Bool} \rangle$, the substitution $\{\alpha \mapsto A\langle \texttt{Int}, \texttt{Bool} \rangle\}$ is returned, and π is updated to $\top$. Second, if M contains variation, the result is computed via case (d). For example, the problem $\alpha \approx^{?} A\langle\star,\texttt{Int}\rangle$ is transformed into $A\langle\alpha,\alpha\rangle \approx^{?} A\langle\star,\texttt{Int}\rangle$.

Next, if M is a function type that contains $\star$ and α does not occur in M, then we transform α into a function type by using fresh type variables and delegate the solving to case (f). The problem (i) in Section 7.1 falls in this case. This case essentially solves two constraints, and we will have two typing patterns (π_1 and π_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.

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

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 $A\langle\star,\alpha\rangle \approx^{?} A\langle\texttt{Int},\texttt{Bool}\rangle$. This constraint will be decomposed into $\star \approx^{?} \texttt{Int}$ and $\alpha \approx^{?} \texttt{Bool}$, 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 $A\langle\star,\texttt{Int}\rangle \approx^{?} \texttt{Int}$, the two smaller constraints to be solved are $\star \approx^{?} \texttt{Int}$ and $\texttt{Int} \approx^{?} \texttt{Int}$. 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 $d\langle\pi_1,\pi_2\rangle$, we check if π_1 and π_2 are the same; if so, the choice pattern is replaced by π_1. In the last two cases of $\sqcap$ 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 $\mathscr{U}$ prevents dead alternatives from being introduced.

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

The first constraint is $A\langle\varepsilon, C_1 \wedge C_2 \wedge C_4\rangle$. 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 ε, which will be solved by case (g) with the solution $(\theta_l, \top)$, where $\theta_l = \varnothing$. The right alternative has the constraint $C_1 \wedge C_2 \wedge C_4$. We will repeatedly use case (i) to handle each subconstraint C_1 through C_4. Since there are no $\star$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

$$\theta_r = \{\alpha \mapsto \texttt{Bool} \to \texttt{Int}, \kappa_1 \mapsto \texttt{Bool}, \kappa_2 \mapsto \texttt{Int}, \kappa_4 \mapsto \texttt{Int}\}$$

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 $A\langle\top,\top\rangle$, which simplifies to $\top$, meaning that the type inference succeeds everywhere. Next, we combine unifiers with the function *merge* defined earlier in this subsection. Note, since θ_l is $\varnothing$, the second case of *merge* will handle each mapping in θ_r. For example, as $\alpha \mapsto \texttt{Bool} \to \texttt{Int} \in \theta_r$, then the merged substitution includes $\alpha \mapsto A\langle\kappa_8, \texttt{Bool} \to \texttt{Int}\rangle$, where κ_8 is s fresh type variable. Here we use a fresh type variable in the first alternative to denote that the first alternative for α is not constrained yet, allowing future unification with any type, if necessary. Overall, let θ_m be the substitution after merging θ_l and θ_r, then

$$\theta_m = \{\alpha \mapsto A\langle\kappa_8, \texttt{Bool} \to \texttt{Int}\rangle, \kappa_1 \mapsto A\langle\kappa_9, \texttt{Bool}\rangle, \kappa_2 \mapsto A\langle\kappa_{10}, \texttt{Int}\rangle, \kappa_4 \mapsto A\langle\kappa_{12}, \texttt{Int}\rangle\}$$

Substituting the result type $A\langle\star,\kappa_2\rangle \to \texttt{Int}$ with θ_m yields the type $A\langle\star, A\langle\kappa_8, \texttt{Bool} \to \texttt{Int}\rangle\rangle \to \texttt{Int}$, which simplifies to the type $A\langle\star, \texttt{Bool} \to \texttt{Int}\rangle \to \texttt{Int}$ after we eliminate the unreachable alternative κ_8. 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 $A\langle\star, \texttt{Bool} \to \texttt{Int}\rangle \to \texttt{Int}$, we know the migrated expression has the type $(\texttt{Bool} \to \texttt{Int}) \to \texttt{Int}$.

Now we solve the constraint $A\langle\varepsilon, C_1 \wedge C_2 \wedge C_4 \wedge C_5 \wedge C_6\rangle$ generated for the expression $\lambda x{:}\star.x\,(\texttt{succ}\ (x\ \texttt{True}))$. We proceed similarly as before. In particular, constraint solving C_1 through C_4 yields the unifier θ_r mentioned above. We then need to solve C_5 and C_6 from θ_r. When solving C_6, we need to unify $\texttt{Bool} \to \texttt{Int}$ with $\texttt{Int} \to \kappa_8$, which fails. The pattern returned is thus $\bot$. Therefore, the pattern for solving the whole constraint is $A\langle\top,\bot\rangle$. 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 $\star$ could actually be assigned to x, such as $\star \to \texttt{Int}$. The reason we cannot find this migration is that $\lambda x.x\,(\texttt{succ}\ (x\ \texttt{True}))$ 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 $\mathscr{U}$ to infer more precise types in Section 9.2.

7.3 Properties

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

Theorem 13 (*Termination*)
Given C, $\mathscr{U}(C)$ terminates.

Next, we show that $\mathscr{U}$ 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, $d\langle M_{11} \approx^? M_{12}, M_{21} \approx^? M_{22}\rangle$ can be transformed into $d\langle M_{11}, M_{21}\rangle \approx^? d\langle M_{12}, M_{22}\rangle$. Note that π in the constraint is just a placeholder and will be updated when the constraint solving finishes.

Theorem 14 (*Soundness*)

If $\mathscr{U}(M_1 \approx^? M_2) = (\theta, \pi')$, 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 $\mathscr{U}(M_1 \approx^? M_2) = (\theta_2, \pi_2)$ such that $\pi_1 \leq \pi_2$ and $\theta_1 = \theta \circ \theta_2$ for some θ.

The idea of the proof is to go through all possible constructs of the type M and show that $\mathscr{U}$ 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 `rowAtISt`, which shares the body with `rowAtI` but removes $\star$s from all its parameters.

```
rowAtISt 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 `width` requires `widthFunc` to have the type `Bool` $\rightarrow$ `Int` and the else-branch requires it to have the type `Int` $\rightarrow$ `Int`.

The difficulties in adding $\star$s are similar to the ones espoused for removing $\star$s in Section 1.1. There is an exponential number of ways $\star$s can be added to the program; adding $\star$s to all parameters introduces more dynamism than desired. Some dynamism can be avoided by adding $\star$ annotations in a left to right manner, but this is inefficient and can still add unnecessary dynamism. For example, following this process on `rowAtISt` leads to a migration that add $\star$s from `headOrFoot` to `border`, since only then `rowAtISt` becomes well typed. In fact, however, the dynamism on, for example, `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 `rowAtISt` can be thought of as one of the programs in the migration space of `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 $\star$s for `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 $\star$s and not to others. Based on the duality, we should now introduce variations to parameters *without* $\star$ annotations and not to others. Specifically, we define a new type system using the judgment form $\pi;\Gamma \vdash_D e : M \,|\, \Omega$. This judgment has the same meaning as the one in Figure 10 and shares the same rules as that one except for Abs and AbsDyn, for which typing rules are as follows.

$$\textsc{Abs}\ \frac{\pi;\Gamma, x \mapsto d\langle\star, V\rangle \vdash_D e : M \,|\, \Omega \qquad d\ \textit{fresh}}{\pi;\Gamma \vdash_D \lambda x.e : d\langle\star, V\rangle \rightarrow M \,|\, \Omega \cup \{x \mapsto d\langle\star, V\rangle\}}$$

$$\textsc{AbsDyn}\ \frac{\pi;\Gamma, x \mapsto \star \vdash_D e : M \,|\, \Omega}{\pi;\Gamma \vdash_D \lambda x : \star.e : \star \rightarrow M \,|\, \Omega}$$

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 $\star$, 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.

$$A\langle\star,\texttt{Bool}\rangle\to B\langle\star,\texttt{Bool}\rangle\to D\langle\star,(\texttt{Int}\to\texttt{Int})\rangle\to E\langle\star,[[\texttt{Char}]]\rangle\to F\langle\star,\alpha\rangle\to G\langle\star,\texttt{Int}\rangle\to[\texttt{Char}]$$

The typing pattern for it is:

$$\pi_d = B\langle F\langle\top,\bot\rangle, D\langle F\langle\top,\bot\rangle,\bot\rangle\rangle$$

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.

$$\Omega[\delta] = \{x\mapsto\lfloor M\rfloor_\delta \mid x\mapsto M\in\Omega\}$$

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.

$$\textsc{Abs}\ \frac{\omega;\Gamma,x\mapsto\omega(x)\vdash_{GCD} e:G}{\omega;\Gamma\vdash_{GCD}\lambda x.e:\omega(x)\to G}\qquad\qquad \textsc{AbsDyn}\ \frac{\omega;\Gamma,x\mapsto\star\vdash_{GCD} e:G}{\omega;\Gamma\vdash_{GCD}\lambda x:\star.e:\star\to G}$$

In Abs, a parameter with a static type is maybe assigned a $\star$ if the ω specifies so. For functions with $\star$ 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 Γ assume e cannot be typed in ITGL under Γ. Let $\pi;\Gamma\vdash_D e:M\,|\,\Omega$. If $\lfloor\pi\rfloor_\delta{=}\top$, 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 $\pi;\Gamma\vdash_D e:M\,|\,\Omega$ where $\lfloor M\rfloor_\delta{=}G$ and $\Omega[\delta]$ for some decision δ.

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 Γ, there is a MGDM typing $\pi;\Gamma \vdash_D e : M \,|\, \Omega$ such that for any $\pi;\Gamma \vdash_D e : M_1 \,|\, \Omega_1$ we have $\forall \delta. \lfloor \pi_1 \rfloor_\delta = \top \Rightarrow \lfloor \pi \rfloor_\delta = \top \wedge \lfloor M_1 \rfloor_\delta \preceq \lfloor M \rfloor_\delta$.

From the typing pattern π 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 π for the MGDM of `rowAtISt` is π_d given earlier. This pattern indicates that either `fixed` and `border` should have $\star$s to remove the type error, or `widthFunc` and `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 AbsDyn rule should be the same as the one in Figure 10 while Abs be the same to the one in $\vdash_D$. After that, we can use the method descried 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

G is a complex type containing $\star$ types, replace G by a choice whose first alternative is G and whose second alternative replaces all dynamic parts by arbitrary types. For example, if $G = \texttt{Int} \to \star \to \star$, then the type of the parameter is $d\langle \texttt{Int} \to \star \to \star, \texttt{Int} \to V_1 \to V_2 \rangle$, where d is fresh. To generate the corresponding constraint (Section 6), we replace V_1 and V_2 by fresh type variables. Note that this extension still tries to assigns full static types for $\star$s. As such, this extension will not be find a migration for $\lambda x{:}\star.x(\texttt{succ}\ (x\ \texttt{True}))$, 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.

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

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

Finally, we can also add support for let-polymorphism. The approach is straightforward, but the notations become heavier. We use $\overline{\alpha}$ to denote a list of type variables and $\{\overline{\alpha \mapsto V}\}$ to denote a set that includes $\alpha_1 \mapsto V_1, \ldots, \alpha_n \mapsto V_n$. The function $vars(\cdot)$ returns the free type variables in its argument. The typing rules are standard except that when typing variable references (VAR) we can only instantiate type schemas with variational types (V) and not migrational types (M).

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

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

9.2 Inferring More Precise Types

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

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 $C_1 \wedge C_2 \wedge C_4 \wedge C_5 \wedge C_6$ below. The first column lists the constraint being solved and the latter two columns list the unifier and pattern from solving the constraint.

Constraint	Solution	Pattern
$\alpha \approx^? \kappa_1 \to \kappa_2$	$\{\alpha \mapsto \kappa_1 \to \kappa_2\}$	$\top$
$\alpha \approx^? \texttt{Bool} \to \kappa_4$	$\{\alpha \mapsto \texttt{Bool} \to \kappa_4, \kappa_1 \mapsto \texttt{Bool}, \kappa_2 \mapsto \kappa_4\}$	$\top$
$\texttt{Int} \approx^? \kappa_2$	$\{\alpha \mapsto \texttt{Bool} \to \texttt{Int}, \kappa_1 \mapsto \texttt{Bool}, \kappa_2 \mapsto \texttt{Int}\}$	$\top$
$\alpha \approx^? \kappa_5 \to \kappa_6$	Ignored, does not affect result	
$\alpha \approx^? \texttt{Int} \to \kappa_8$	$\{\alpha \mapsto \texttt{Bool} \to \texttt{Int}, \kappa_1 \mapsto \texttt{Bool}, \kappa_2 \mapsto \texttt{Int}\}$	$\bot$

The constraint solving fails when we need to solve the constraint $\alpha \approx^? \texttt{Int} \to \kappa_8$, since our solution before that point contains $\alpha \mapsto \texttt{Bool} \to \texttt{Int}$. When constraint solving fails, the returned pattern is $\bot$, and the content of the unifier will no longer be used. As a result, we leave the content of the unifier as the same after solving $\alpha \approx^? \texttt{Int} \to \kappa_8$.

The main reason our approach fails to find a migration is that, as we were solving the first constraint $\alpha \approx^? \kappa_1 \to \kappa_2$, we made three requirements: 1) the type that α 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 $x(\texttt{succ}\ (x\ \texttt{True}))$, the body of the function, x is used as functions and applied to both `Bool` and `Int` values. As a result, no static type could be assigned to x. We can address this problem by relaxing the three requirements for α. To address this problem, we observe that α denotes the type for x when the $\star$ for x is removed, and we are finding a more precise migration than $\star$. Thus, instead of constraining α with all the three requirements at once, we can relax the latter two requirements and require α be unified with a type whose type constructor is $\to$ only. From now on, we call type variables that are introduced to replace $\star$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 α 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 $\star$. The typing that happens later decides whether the parameter type and/or return type could be made precise than a $\star$. As a result, a parameter can now be migrated to a function type whose parameter or return type remains a $\star$.

One technical challenge is that for the parameter type and return type, we need to explore two possibilities: the $\star$ and a more precise type. Our machinery with variational typing provides a nice solution. Specifically, when a migration variable α is unified with a function type $M_1 \to M_2$, we refine α to a function type $A_1\langle\star, \alpha_1\rangle \to A_2\langle\star, \alpha_2\rangle$ (We refer to this process as *refinement*) and unify this function type against $M_1 \to M_2$. Here, A_1, α_1, A_2, and α_2 are fresh and α_1 and α_2 are migration variables, which could be further refined to function types whose parameter and return types are $\star$s. The function type $A_1\langle\star, \alpha_1\rangle \to A_2\langle\star, \alpha_2\rangle$ encodes four possibilities: both the parameter type and the return type could be $\star$ 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 $\alpha_1 \mapsto \kappa_1$ and $\alpha_2 \mapsto \kappa_2$ to save space.

(bR) $\mathscr{U}(\beta \approx^? M)$
  $\mid \beta \notin vars(M) \wedge \neg hasDyn(M) = (\{\beta \mapsto M\}, \top)$
  $\mid d \in choices(M) = \mathscr{U}(d\langle\beta,\beta\rangle \approx^? M)$
  $\mid \beta \notin vars(M) \wedge M$ is of form $M_1 \to M_2 =$
    let $(\theta_1,\pi_1) = \mathscr{U}(\beta \approx^? \kappa_1 \to \kappa_2)$; $(\theta_2,\pi_2) = \mathscr{U}(\kappa_1 \to \kappa_2 \approx^? M_1 \to M_2)$ in $(\theta_2 \circ \theta_1, \pi_2 \sqcap \pi_1)$
  $\mid$ otherwise $= (\emptyset, \bot)$
(bR*) $\mathscr{U}(M \approx^? \beta) = \mathscr{U}(\beta \approx^? M)$
(b1) $\mathscr{U}(\alpha \approx^? \alpha) = (\varnothing, \top)$
(b2) $\mathscr{U}(\alpha \approx^? \gamma) = (\{\alpha \mapsto \gamma\}, \top)$
(b3) $\mathscr{U}(\alpha \approx^? \beta) = (\{\alpha \mapsto \beta\}, \top)$
(b4) $\mathscr{U}(\alpha \approx^? d\langle M_1,M_2\rangle) = \mathscr{U}(d\langle\alpha,\alpha\rangle \approx^? d\langle M_1,M_2\rangle)$
(b5) $\mathscr{U}(\alpha \approx^? M_1 \to M_2)$
  $\mid AllLvsDynMvs(M_1 \to M_2) \wedge \alpha \in vars(M_1 \to M_2) = (\varnothing, \bot)$
  $\mid AllLvsDynMvs(M_1 \to M_2) \wedge \neg hasDyn\ (M_1 \to M_2) = (\{\alpha \mapsto M_1 \to M_2\}, \top)$
  $\mid AllLvsDynMvs(M_1 \to M_2) =$
    let $(\theta_1,\pi_1) = \mathscr{U}(\beta \approx^? \kappa_1 \to \kappa_2)$; $(\theta_2,\pi_2) = \mathscr{U}(\kappa_1 \to \kappa_2 \approx^? M_1 \to M_2)$ in $(\theta_2 \circ \theta_1, \pi_2 \sqcap \pi_1)$
  $\mid$ otherwise $=$
    let $\theta_1 = \{\alpha \mapsto A_1\langle\star,\alpha_1\rangle \to A_2\langle\star,\alpha_2\rangle\}$ A_1, A_2, α_1, and α_2 fresh
      $(\theta_2,\pi_2) = \mathscr{U}(A_1\langle\star,\alpha_1\rangle \to A_2\langle\star,\alpha_2\rangle \approx^? \theta_1(M_1 \to M_2))$
    in $(\theta_2 \circ \theta_1, \pi_2)$
(b6) $\mathscr{U}(M \approx^? \alpha) = \mathscr{U}(\alpha \approx^? M)$

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

Constraint	Solution	Pattern
$\alpha \approx^? \kappa_1 \to \kappa_2$	$\{\alpha \mapsto A_1\langle\star,\kappa_1\rangle \to A_2\langle\star,\kappa_2\rangle\}$	$\top$
$\alpha \approx^? \texttt{Bool} \to \kappa_4$	$\{\alpha \mapsto A_1\langle\star,\texttt{Bool}\rangle \to A_2\langle\star,\kappa_4\rangle, \kappa_1 \mapsto \texttt{Bool}, \kappa_2 \mapsto \kappa_4\}$	$\top$
$\texttt{Int} \approx^? \kappa_2$	$\{\alpha \mapsto A_1\langle\star,\texttt{Bool}\rangle \to A_2\langle\star,\texttt{Int}\rangle, \kappa_1 \mapsto \texttt{Bool}, \kappa_2 \mapsto \texttt{Int}\}$	$\top$
$\alpha \approx^? \kappa_5 \to \kappa_6$	$\{\alpha \mapsto A_1\langle\star,\texttt{Bool}\rangle \to A_2\langle\star,\texttt{Int}\rangle, \kappa_1 \mapsto \texttt{Bool}, \kappa_2 \mapsto \texttt{Int},$ $\kappa_5 \mapsto A_1\langle\kappa_9,\texttt{Bool}\rangle, \kappa_6 \mapsto A_2\langle\kappa_{10},\texttt{Int}\rangle\ \}$	$\top$
$\alpha \approx^? \texttt{Int} \to \kappa_8$	Extend above with $\{\ \kappa_8 \mapsto A_2\langle\kappa_{12},\texttt{Int}\rangle\ \}$	$A_1\langle\top,\bot\rangle$

From Section 6 (page 32), we know that the type of $\lambda x\!:\!\star.x(\texttt{succ}\ (x\ \texttt{True}))$ is $A\langle\star,\alpha\rangle \to A\langle\star,\kappa_6\rangle$. Plugging in the solution for α from the unifier above, the type for $\lambda x\!:\!\star.x(\texttt{succ}\ (x\ \texttt{True}))$ is $M_{dp} = A\langle\star, A_1\langle\star,\texttt{Bool}\rangle \to A_2\langle\star,\texttt{Int}\rangle\rangle \to A\langle\star, A_2\langle\kappa_{10},\texttt{Int}\rangle\rangle$. Moreover, the pattern for the whole function is $A\langle\top, A_1\langle\top,\bot\rangle\rangle$. Note, A_2 does not appear in the result pattern because whether we choose $\star$ or `Int` for the return type of the function type for α, the well-typedness of the expression remains the same. Applying the operations *ve* and *expand*, defined in Section 5.2, to the pattern $A\langle\top, A_1\langle\top,\bot\rangle\rangle$, we know that the best migration for this expression corresponds to the valid eliminator $\{A.2, A_1.1, A_2.2\}$. Selecting M_{dp} with $\{A.2, A_1.1, A_2.2\}$ yields the type $(\star \to \texttt{Int}) \to \texttt{Int}$, the type of $\lambda x\!:\!\star.x(\texttt{succ}\ (x\ \texttt{True}))$ after migrating the parameter x. This means that our extension could indeed find a more precise migration for $\lambda x\!:\!\star.x(\texttt{succ}\ (x\ \texttt{True}))$.

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 α to β. Note that from now on, we use α to denote migration variables and β 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 *AllLvsDynMvs* to determine if the leaves of a given input type are all $\star$s or migration type variables. For example, all $\mathit{AllLvsDynMvs}\,(\alpha_1 \to \alpha)$, $\mathit{AllLvsDynMvs}\,(\alpha_2)$, and $\mathit{AllLvsDynMvs}\,((\star \to \alpha) \to \alpha_2)$ are true, while $\mathit{AllLvsDynMvs}\,(\alpha_1 \to \texttt{Int})$ and $\mathit{AllLvsDynMvs}\,((\alpha_1 \to \texttt{Bool}) \to \alpha_2)$ are false. This function helps avoid non-termination in our extension. To illustrate, consider the constraint $\alpha \approx^? \alpha \to \beta$. Such a constraint arises when typing a self application, such as in the expression $\lambda x{:}\star.x\ x$. 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 $\alpha \approx^? \alpha \to \beta$ into $A_1\langle\star,\alpha_1\rangle \to A_2\langle\star,\alpha_2\rangle \approx^? (A_1\langle\star,\alpha_1\rangle \to A_2\langle\star,\alpha_2\rangle) \to \beta$. This constraint encodes four constraints, and one of them is $\alpha_1 \to \alpha_2 \approx^? (\alpha_1 \to \alpha_2) \to \beta$ (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 $\alpha \approx^? \alpha \to \beta$ and the constraint between the parameter types ($\alpha_1 \approx^? \alpha_1 \to \alpha_2$) 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 (α) against a function type that contains the migration variable (α) and all other leaves are other migration variables or $\star$s. Such a unification does not reflect any program structure information but is resulted from refining a unification 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 $\alpha_1 \to \alpha_2 \approx^? (\alpha_1 \to \alpha_2) \to \beta$, which means the typing pattern returned for unifying it will be $\bot$, the typing pattern for unifying $A_1\langle\star,\alpha_1\rangle \to A_2\langle\star,\alpha_2\rangle \approx^? (A_1\langle\star,\alpha_1\rangle \to A_2\langle\star,\alpha_2\rangle) \to \beta$ will not be $\bot$. It is $A_1\langle\top,\bot\rangle$. This means that the pattern for solving $\alpha \approx^? \alpha \to \beta$ 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, α does not occur in the function type and all leaves are migration variables, then we directly map α to the function type. In the third subcase, the function type contains some $\star$s. We need to refine α 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 $\lambda x{:}\star.x\ x$. First, we generate the constraint $A\langle\star,\alpha\rangle \approx^? A\langle\star,\alpha\rangle \to \beta$ and the type for the term is $A\langle\star,\alpha\rangle \to \beta$. 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 $\star \approx^? \star \to \beta$, which will be solved by case (a) of Figure 13 with the solution $(\varnothing, \top)$. For the right alternative, the constraint is $\alpha \approx^? \alpha \to \beta$. This constraint will be handled by the fourth subcase of case (b5) in Figure 14, and it will be transformed to $A_1\langle\star,\alpha_1\rangle \rightarrow A_2\langle\star,\alpha_2\rangle \approx^? (A_1\langle\star,\alpha_1\rangle \rightarrow A_2\langle\star,\alpha_2\rangle) \rightarrow \beta$.

With a few steps, this problem can be solved and the solution is $\{\alpha \mapsto A_1\langle\star,\alpha_1\rangle \rightarrow A_2\langle\star,\beta\rangle, \alpha_2 \mapsto \beta\}$ and the pattern is $A_1\langle\top,\bot\rangle$. Substituting the type of the term with this solution yields $A\langle\star, A_1\langle\star,\alpha_1\rangle \rightarrow A_2\langle\star,\beta\rangle\rangle \rightarrow \beta$ and the overall pattern is $A\langle\top, A_1\langle\top,\bot\rangle\rangle$. From this pattern, we can use *ve* and *expand* defined in Section 5.2 to calculate the strictest valid eliminator $\{A.2, A_1.1, A_2.2\}$. Selecting the type $A\langle\star, A_1\langle\star,\alpha_1\rangle \rightarrow A_2\langle\star,\beta\rangle\rangle \rightarrow \beta$ with this eliminator leads to the type $(\star \rightarrow \beta) \rightarrow \beta$, which is a most static migration for $\lambda x\!:\!\star . x\ x$. This shows that with the extended constraint solving algorithm, we could find a more precise migration for $\lambda x\!:\!\star . x\ x$ 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.

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

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 δ_r for *refining* the pattern π_a based on this requirement. To express that `table` must be migrated, we extend δ_r with $D.2$, as D is the variation introduced for `table`. For `widthFunc` to be not migrated, we extend δ_r with $B.1$, making $\delta_r = \{B.1, D.2\}$. After that, we refine π_a with δ_r, yielding the new pattern $A\langle E\langle\top,\bot\rangle, E\langle\top,\bot\rangle\rangle$, which could be simplified to $E\langle\top,\bot\rangle$. We can now apply the method developed in Section 5 to the pattern $E\langle\top,\bot\rangle$ to find the best migrations for `rowAtI` while honoring the requirements. Based on the pattern $E\langle\top,\bot\rangle$, 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 π generated from MGSM, we follow the following steps to handle this scenario.

(1) For each parameter that must be migrated, we extend δ_r with $d.2$, where d is the variation introduced for the parameter.
(2) For each parameter that must not be migrated, we extend δ_r with $d.1$, where d is the variation introduced for the parameter.
(3) We refine the pattern π with δ_r.
(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 `fixed`, `widthFunc`, and `table`. According to the process of calculating δ_r given earlier, $\delta_r = \{A.2, B.2, D.2\}$. We observe that $\lfloor \pi_a \rfloor_{\delta_r} = \bot$, 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 `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 `fixed` (since `fixed` corresponds to A and $A.2$ belongs to the set) and `table` of the desired set could be migrated. From the second set, we could decide that `widthFunc` and `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 `fixed` and `border`, then the unique migration corresponds to the decision is $\{A.2, B.1, D.2, E.1, F.2\}$, indicating that only `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 `rowAtI`, we could not migrate `widthFunc` if we have migrated `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 `fixed` and that we want to calculate the impact it has on other parameters. We must now consider two cases. The first case migrates `fixed`, and the decision is $\delta_r = \{A.2\}$. The second case does not migrate `fixed`, and the decision is $\delta_{\neg r} = \{A.1\}$. Let π_r and $\pi_{\neg r}$ denote the typing patterns resulted from selecting π_a with δ_r and $\delta_{\neg r}$, respectively, we have

$$\pi_r = B\langle E\langle \top, \bot \rangle, \bot \rangle \qquad \pi_{\neg r} = E\langle \top, \bot \rangle$$

In the first case, from π_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 `border`, the parameter corresponding to E, always causes an error, meaning that `fixed` being migrated was irrelevant to the reason `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 `widthFunc` can not be migrated is because `fixed` was migrated. Consequently, the parameter that can not be migrated in unison with `fixed` is `widthFunc`.

Given an expression e and π 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.

(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 π with π_r.
(3) Collect the decisions that produce $\bot$ when selecting π with $\pi_{\neg r}$.
(4) For any d', if $d'.2$ appears in some decisions from step (3) but not from any of 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 most 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 most static migrations for `rowAtI` migrate three parameters and both are maximal.

Given an expression and its typing pattern π for its MGSM, a simple process to find maximal migrations is generate all best migrations from π 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*.

$$
\begin{aligned}
mve(\top) &= \{\varnothing\} \\
mve(\bot) &= \varnothing \\
mve(d\langle \pi_1, \pi_2 \rangle) &= \begin{cases} lmve & rmve = \varnothing \text{ or } |\mathscr{D}| - |lmve[0]|_1| > |\mathscr{D}| - |rmve[0]|_1| \\ rmve & |\mathscr{D}| - |lmve[0]|_1| < |\mathscr{D}| - |rmve[0]|_1| \\ lmve \cup rmve & \text{otherwise} \end{cases} \\
& \quad \text{where } lmve = \{\{d.1\} \cup l \mid l \in mve(\pi_1)\} \\
& \quad\quad\quad\; rmve = \{\{d.2\} \cup r \mid r \in mve(\pi_2)\}
\end{aligned}
$$

In the definition, $\delta|_1$ (introduced in Section 4.5) returns all left selectors in δ. The notation $lmve[0]$ returns any member from the set $lmve$. This is valid because all of the members in $lmve$ include the same number of left selectors, and so do those in $rmve$. The set $\mathscr{D}$ (introduced in Section 5.2) contains all variations introduced in typing e. Note, given a decision δ, if $d.1 \notin \delta$ then the parameter corresponding to d can not be migrated. Therefore, $|\mathscr{D}| - |lmve[0]|_1|$ gives the number of parameters that can be migrated in $lmve[0]$.

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

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

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.

scenarios (i) and (iv) could be supported by following the first three steps outlined in Scenario (i) and then applying the *mve* function to the resulting pattern. As another example, we may be interested in the scenario of finding the maximal migration within a given set of parameters. To support this scenario, we first select the typing pattern of the MGSM typing with selectors of the form $d.1$, where d corresponds to a parameter that does not belong to the given set. The selection result is a pattern, to which we apply *mve* to find the maximal migration within that parameter set.

Overall, the generality of our approach demonstrates that it could be a useful foundation for developing more complex and significant migration supports in practice.

10 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

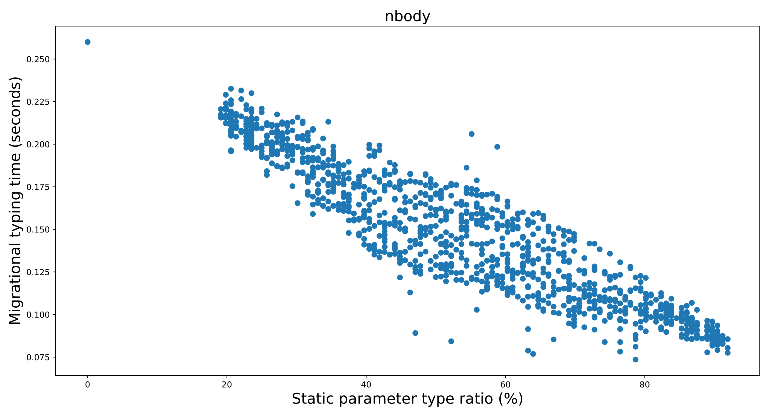

Fig. 16: Relations between ratios of typed parameters and migrational typing times for the nbody benchmark.

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 $\star$ 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 `Bool` and `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 $\star$ with a more precise type and tests the well-typedness of the generated program. They find a maximal migration if no more $\star$s exist of no more $\star$ could be replaced with any more precise type. A migration in our work is maximal if no further $\star$ 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 $\lambda x:\star.x\ (succ\ (x\ \texttt{True}))$, their approach infers that x can be given the type $\star \rightarrow \texttt{Int}$, 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 $\star \rightarrow \texttt{Int}$).

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.

```
width fixed widthFunc = 2 + (if fixed then widthFunc fixed else widthFunc 33)
```

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

```
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 $\star$ whereas the type for `widthFunc` is `Int-> Int`, as shown below.

```
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 $\star$). The migrated program is shown below.

```
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 $\star$ 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 $\star$ to `fixed` that has the type `Int` so that `fixed` can be used where a `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 `width` function accepts only `Int` values, which will lead to a runtime error since `fixed` is used as a `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 can not 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 & Taha, 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 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 Kazerounian *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 Kazerounian *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 $A\langle\bot, B\langle\top, C\langle\bot,\top\rangle\rangle\rangle$ has three internal nodes and four leaves, while an equivalent pattern $C\langle A\langle\bot, B\langle\top,\bot\rangle\rangle, A\langle\bot,\top\rangle\rangle$ 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 $\star$s when they are created and $\star$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, $A\langle\top, B\langle\top, C\langle\top,\bot\rangle\rangle\rangle$ has seven nodes, and $C\langle\top, A\langle\top, B\langle\top,\bot\rangle\rangle\rangle$, 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

Ahmed, Amal, Findler, Robert Bruce, Siek, Jeremy G., & Wadler, Philip. (2011). Blame for all. *Sigplan not.*, **46**(1), 201–214.

Ahmed, Amal, Jamner, Dustin, Siek, Jeremy G., & Wadler, Philip. (2017). Theorems for free for free: Parametricity, with and without types. *Proc. acm program. lang.*, **1**(ICFP), 39:1–39:28.

Akers, S. B. (1978). Binary decision diagrams. *Ieee transactions on computers*, **C-27**(6), 509–516.

Allende, Esteban, Fabry, Johan, Garcia, Ronald, & Tanter, Éric. (2014). Confined gradual typing. *Sigplan not.*, **49**(10), 251–270.

Apel, Sven, Batory, Don, Kästner, Christian, & Saake, Gunter. (2016). *Feature-oriented software product lines*. Springer.

Bayne, Michael, Cook, Richard, & Ernst, Michael D. (2011). Always-available static and dynamic feedback. *Pages 521–530 of: Proceedings of the 33rd International Conference on Software Engineering*. ICSE '11. New York, NY, USA: ACM.

Bryant, Randal E. (1992). Symbolic boolean manipulation with ordered binary-decision diagrams. *Acm comput. surv.*, **24**(3), 293–318.

Campora, John, Chen, Sheng, Erwig, Martin, & Walkingshaw, Eric. (2018a). Migrating gradual types. *Proceedings of the 45th ACM SIGPLAN Symposium on Principles of Programming Languages*. POPL '18. New York, NY, USA: ACM.

Campora, John, Chen, Sheng, Erwig, Martin, & Walkingshaw, Eric. (2022). *Migrating gradual types*. Tech. rept. University of Louisiana at Lafayette. https://people.cmix.louisiana.edu/schen/ws/techreport/MGT-With-Proofs.pdf.

Campora, John Peter, & Chen, Sheng. (2020). Taming type annotations in gradual typing. *Proc. acm program. lang.*, **4**(OOPSLA).

Campora, John Peter, Chen, Sheng, & Walkingshaw, Eric. (2018b). Casts and costs: Harmonizing safety and performance in gradual typing. *Proc. acm program. lang.*, **2**(ICFP), 98:1–98:30.

Castagna, Giuseppe, & Lanvin, Victor. (2017). Gradual typing with union and intersection types. *Proc. acm program. lang.*, **1**(ICFP), 41:1–41:28.

Castagna, Giuseppe, Lanvin, Victor, Petrucciani, Tommaso, & Siek, Jeremy G. (2019). Gradual typing: A new perspective. *Proc. acm program. lang.*, **3**(POPL).

Chaudhuri, Avik, Vekris, Panagiotis, Goldman, Sam, Roch, Marshall, & Levi, Gabriel. (2017). Fast and precise type checking for javascript. *Proc. acm program. lang.*, **1**(OOPSLA), 48:1–48:30.

Chen, Sheng, & Campora III, John Peter. (2019). Blame Tracking and Type Error Debugging. *Pages 2:1–2:14 of:* Lerner, Benjamin S., Bodík, Rastislav, & Krishnamurthi, Shriram (eds), *3rd Summit on Advances in Programming Languages (SNAPL 2019)*.

Leibniz International Proceedings in Informatics (LIPIcs), vol. 136. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

Chen, Sheng, & Erwig, Martin. (2014a). Counter-factual typing for debugging type errors. *Pages 583–594 of: Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '14. New York, NY, USA: ACM.

Chen, Sheng, & Erwig, Martin. (2014b). Guided type debugging. *Pages 35–51 of: Int. Symp. on Functional and Logic Programming*. LNCS 8475.

Chen, Sheng, & Erwig, Martin. (2016). Principal type inference for gadts. *Pages 416–428 of: Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '16. New York, NY, USA: ACM.

Chen, Sheng, Erwig, Martin, & Walkingshaw, Eric. (2012). An error-tolerant type system for variational lambda calculus. *Pages 29–40 of: Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming*. ICFP '12. New York, NY, USA: ACM.

Chen, Sheng, Erwig, Martin, & Walkingshaw, Eric. (2014). Extending type inference to variational programs. *Acm trans. program. lang. syst.*, **36**(1), 1:1–1:54.

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

Chugh, Ravi, Rondon, Patrick M., & Jhala, Ranjit. (2012). Nested refinements: A logic for duck typing. *Page 231–244 of: Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '12. New York, NY, USA: Association for Computing Machinery.

Cimini, Matteo, & Siek, Jeremy G. (2016). The gradualizer: A methodology and algorithm for generating gradual type systems. *Pages 443–455 of: Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '16. New York, NY, USA: ACM.

Cimini, Matteo, & Siek, Jeremy G. (2017). Automatically generating the dynamic semantics of gradually typed languages. *Pages 789–803 of: Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*. POPL 2017. New York, NY, USA: ACM.

Danner, Norman, Licata, Daniel R., & Ramyaa, Ramyaa. (2015). Denotational cost semantics for functional languages with inductive types. *Pages 140–151 of: Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*. ICFP 2015. New York, NY, USA: ACM.

Erwig, Martin, & Walkingshaw, Eric. (2011). The choice calculus: A representation for software variation. *Acm trans. softw. eng. methodol.*, **21**(1), 6:1–6:27.

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

Garcia, Ronald, & Cimini, Matteo. (2015). Principal type schemes for gradual programs. *Pages 303–315 of: Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '15. New York, NY, USA: ACM.

Garcia, Ronald, Clark, Alison M., & Tanter, Éric. (2016). Abstracting gradual typing. *Pages 429–442 of: Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '16. New York, NY, USA: ACM.

Guha, Arjun, Matthews, Jacob, Findler, Robert Bruce, & Krishnamurthi, Shriram. (2007). Relationally-parametric polymorphic contracts. *Page 29–40 of: Proceedings of the 2007 Symposium on Dynamic Languages*. DLS '07. New York, NY, USA: Association for Computing Machinery.

Guha, Arjun, Saftoiu, Claudiu, & Krishnamurthi, Shriram. (2011). Typing local control and state using flow analysis. *Pages 256–275 of:* Barthe, Gilles (ed), *Programming Languages and Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg.

Henglein, Fritz, & Rehof, Jakob. (1995). Safe polymorphic type inference for a dynamically typed language: Translating scheme to ml. *Page 192–203 of: Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture*. FPCA '95. New York, NY, USA: Association for Computing Machinery.

Igarashi, Atsushi, Thiemann, Peter, Vasconcelos, Vasco T., & Wadler, Philip. (2017). Gradual session types. *Proc. acm program. lang.*, **1**(ICFP), 38:1–38:28.

Jafery, Khurram A., & Dunfield, Jana. (2017). Sums of uncertainty: Refinements go gradual. *Pages 804–817 of: Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*. POPL 2017. New York, NY, USA: ACM.

Kazerounian, Milod, Ren, Brianna M., & Foster, Jeffrey S. (2020). *Sound, heuristic type annotation inference for ruby*. New York, NY, USA: Association for Computing Machinery. Page 112–125.

Kazerounian, Milod, Foster, Jeffrey S., & Min, Bonan. (2021). Simtyper: Sound type inference for ruby using type equality prediction. *Proc. acm program. lang.*, **5**(OOPSLA).

Kuhlenschmidt, Andre, Almahallawi, Deyaaeldeen, & Siek, Jeremy G. (2019). Toward efficient gradual typing for structural types via coercions. *Page 517–532 of: Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2019. New York, NY, USA: Association for Computing Machinery.

Lehmann, Nico, & Tanter, Éric. (2017). Gradual refinement types. *Pages 775–788 of: Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*. POPL 2017. New York, NY, USA: ACM.

Loncaric, Calvin, Chandra, Satish, Schlesinger, Cole, & Sridharan, Manu. (2016). A practical framework for type inference error explanation. *Pages 781–799 of: OOPSLA*.

Marceau, Guillaume, Fisler, Kathi, & Krishnamurthi, Shriram. (2011a). Measuring the effectiveness of error messages designed for novice programmers. *Pages 499–504 of: Proceedings of the 42nd ACM technical symposium on Computer science education*. ACM.

Marceau, Guillaume, Fisler, Kathi, & Krishnamurthi, Shriram. (2011b). Mind your language: on novices' interactions with error messages. *Pages 3–18 of: Proceedings of the 10th SIGPLAN symposium on New ideas, new paradigms, and reflections on programming and software*. ACM.

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

Migeed, Zeina, & Palsberg, Jens. (2019). What is decidable about gradual types? *Proc. acm program. lang.*, **4**(POPL).

Miyazaki, Yusuke, Sekiyama, Taro, & Igarashi, Atsushi. (2019). Dynamic type inference for gradual hindley–milner typing. *Proc. acm program. lang.*, **3**(POPL).

Munson, Jonathan P, & Schilling, Elizabeth A. (2016). Analyzing novice programmers' response to compiler error messages. *Journal of computing sciences in colleges*, **31**(3), 53–61.

Neocleous, Costas, & Schizas, Christos. (2002). Artificial neural network learning: A comparative review. *Pages 300–313 of: Hellenic Conference on Artificial Intelligence*. Springer.

New, Max S., Jamner, Dustin, & Ahmed, Amal. (2019). Graduality and parametricity: Together again for the first time. *Proc. acm program. lang.*, **4**(POPL).

Nguyen, Hung Viet, Kästner, Christian, & Nguyen, Tien N. (2014). Exploring Variability-Aware Execution for Testing Plugin-Based Web Applications. *Pages 907–918 of: Int. Conf. on Software Engineering*. ACM.

Pavlinovic, Zvonimir, King, Tim, & Wies, Thomas. (2014). Finding minimum type error sources. *Pages 525–542 of: OOPSLA*.

Pearce, David J. (2013). A calculus for constraint-based flow typing. *Proceedings of the 15th Workshop on Formal Techniques for Java-like Programs*. FTfJP '13. New York, NY, USA: Association for Computing Machinery.

Phipps-Costin, Luna, Anderson, Carolyn Jane, Greenberg, Michael, & Guha, Arjun. (2021). Solver-based gradual type migration. *Proc. acm program. lang.*, **5**(OOPSLA).

Rastogi, Aseem, Chaudhuri, Avik, & Hosmer, Basil. (2012). The ins and outs of gradual type inference. *Page 481–494 of: Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '12. New York, NY, USA: Association for Computing Machinery.

Robinson, J. A. (1965a). A machine-oriented logic based on the resolution principle. *Journal of the acm*, **12**(1), 23–41.

Robinson, J. A. (1965b). A machine-oriented logic based on the resolution principle. *J. acm*, **12**(1), 23–41.

Sergey, Ilya, & Clarke, Dave. (2012). Gradual ownership types. *Pages 579–599 of: Proceedings of the 21st European Conference on Programming Languages and Systems*. ESOP'12. Berlin, Heidelberg: Springer-Verlag.

Serrano, Alejandro, & Hage, Jurriaan. (2016). *Type error diagnosis for embedded dsls by two-stage specialized type rules*. Berlin, Heidelberg: Springer Berlin Heidelberg. Pages 672–698.

Siek, Jeremy, & Taha, Walid. (2007). Gradual typing for objects. *Pages 2–27 of: Proceedings of the 21st European Conference on ECOOP 2007: Object-Oriented Programming*. ECOOP '07. Berlin, Heidelberg: Springer-Verlag.

Siek, Jeremy G., & Taha, Walid. (2006). Gradual typing for functional languages. *Pages 81–92 of: Workshop on Scheme and Functional Programming*.

Siek, Jeremy G., & Tobin-Hochstadt, Sam. (2016). *The recursive union of some gradual types*. Cham: Springer International Publishing. Pages 388–410.

Siek, Jeremy G., & Vachharajani, Manish. (2008). Gradual typing with unification-based inference. *Pages 7:1–7:12 of: Proceedings of the 2008 Symposium on Dynamic Languages*. DLS '08. New York, NY, USA: ACM.

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

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

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

Takikawa, Asumu, Feltey, Daniel, Greenman, Ben, New, Max S., Vitek, Jan, & Felleisen, Matthias. (2016). Is sound gradual typing dead? *Pages 456–468 of: Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '16. New York, NY, USA: ACM.

Tansey, Wesley, & Tilevich, Eli. (2008). Annotation refactoring: Inferring upgrade transformations for legacy applications. *Pages 295–312 of: Proceedings of the 23rd ACM SIGPLAN Conference on Object-oriented Programming Systems Languages and Applications*. OOPSLA '08. New York, NY, USA: ACM.

Thüm, Thomas, Apel, Sven, Kästner, Christian, Schaefer, Ina, & Saake, Gunter. (2014). A classification and survey of analysis strategies for software product lines. **47**(1), 6:1–6:45.

Tobin-Hochstadt, Sam, & Felleisen, Matthias. (2006). Interlanguage migration: From scripts to programs. *Page 964–974 of: Companion to the 21st ACM SIGPLAN Symposium on Object-Oriented Programming Systems, Languages, and Applications*. OOPSLA '06. New York, NY, USA: Association for Computing Machinery.

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

Toro, Matías, & Tanter, Éric. (2017). A gradual interpretation of union types. *SAS*.

Toro, Matías, Labrada, Elizabeth, & Tanter, Éric. (2019). Gradual parametricity, revisited. *Proc. acm program. lang.*, **3**(POPL).

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.

Vytiniotis, Dimitrios, Peyton jones, Simon, Schrijvers, Tom, & Sulzmann, Martin. (2011). Outsidein(x) modular type inference with local assumptions. *J. funct. program.*, **21**(4-5), 333–412.

Vytiniotis, Dimitrios, Peyton Jones, Simon, & Magalhães, José Pedro. (2012). Equality proofs and deferred type errors: a compiler pearl. *Pages 341–352 of: Proceedings of the 17th ACM SIGPLAN international conference on Functional programming*. ICFP '12.

Wadler, Philip, & Findler, Robert Bruce. (2007). Well-typed programs can't be blamed. *Pages 1–13 of: Proceedings of the 2007 Workshop on Scheme and Functional Programming*.

Wadler, Philip, & Findler, Robert Bruce. (2009). Well-typed programs can't be blamed. *Pages 1–16 of: Proceedings of the 18th European Symposium on Programming Languages and Systems: Held As Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009*. ESOP '09. Berlin, Heidelberg: Springer-Verlag.

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

Walkingshaw, Eric, Kästner, Christian, Erwig, Martin, Apel, Sven, & Bodden, Eric. (2014). Variational data structures: Exploring tradeoffs in computing with variability. *Pages 213–226 of: Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*. Onward! 2014. New York, NY, USA: ACM.

Wei, Jiayi, Goyal, Maruth, Durrett, Greg, & Dillig, Isil. (2020). Lambdanet: Probabilistic type inference using graph neural networks. *International Conference on Learning Representations*.

Williams, Jack, Morris, J. Garrett, & Wadler, Philip. (2018). The root cause of blame: Contracts for intersection and union types. *Proc. acm program. lang.*, **2**(OOPSLA).

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, $\lfloor M \rfloor_{d.i} = \lfloor \lfloor M \rfloor_{d.i} \rfloor_{d.i}$.

Lemma 4 (*Selector ordering is irrelevant*)
For any two variations A and B: $\lfloor \lfloor M \rfloor_{B.j} \rfloor_{A.i} = \lfloor \lfloor M \rfloor_{A.i} \rfloor_{B.j}$

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 $\star$s and $\lfloor \star \rfloor_s = \star$. We omit the detailed proof here.

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

Lemma 5 (*Context filling preserves equivalence*)
$\lfloor M_1 \rfloor_\delta \equiv \lfloor M_2 \rfloor_\delta \wedge \lfloor M[M_1] \rfloor_\delta \in V \wedge \lfloor M[M_2] \rfloor_\delta \in V \Rightarrow \lfloor M[M_1] \rfloor_\delta \equiv \lfloor M[M_2] \rfloor_\delta$

Proof
By structural induction of the syntax of type contexts $M[]$.

Case []: From the implication in the lemma, we are given the following.

$$\lfloor M_1 \rfloor_\delta \equiv \lfloor M_2 \rfloor_\delta \qquad \lfloor M_1 \rfloor_\delta \in V \qquad \lfloor M_2 \rfloor_\delta \in V$$

Since filling the context [] with any type yields that type itself, the proof for this case is immediate.

Case $M'[] \to M$: We are given the following relations

$$\lfloor M_1 \rfloor_\delta \equiv \lfloor M_2 \rfloor_\delta \quad \lfloor M'[M_1] \rfloor_\delta \in V \quad \lfloor M'[M_2] \rfloor_\delta \in V$$

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,

$$\lfloor M'[M_1] \to M \rfloor_\delta = \lfloor M'[M_2] \to M \rfloor_\delta$$

which can be transformed to the following based on the definition of selection.

$$\lfloor M'[M_1] \rfloor_\delta \to \lfloor M \rfloor_\delta = \lfloor M'[M_2] \rfloor_\delta \to \lfloor M \rfloor_\delta$$

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

Case $d\langle M'[], M \rangle$: We have the following implicants and the final equivalence by the induction hypothesis:

$$\lfloor M_1 \rfloor_\delta \equiv \lfloor M_2 \rfloor_\delta \quad \lfloor M'[M_1] \rfloor_\delta \in V \quad \lfloor M'[M_2] \rfloor_\delta \in V \quad \lfloor M'[M_1] \rfloor_\delta \equiv \lfloor M'[M_2] \rfloor_\delta$$

We need to show the following,

$$\lfloor d\langle M'[M_1], M\rangle \rfloor_\delta = \lfloor d\langle M'[M_2], M\rangle \rfloor_\delta$$

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,

$$\lfloor M'[M_1] \rfloor_\delta = \lfloor M'[M_2] \rfloor_\delta$$

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,

$$\lfloor M \rfloor_\delta = \lfloor M \rfloor_\delta$$

Thus, the lemma holds for this case.

Case $d\langle M, M'[]\rangle$: 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,

$$M_1 \approx M_2 \quad M[M_1] \approx M[M_2] \quad \lfloor M[M_1] \rfloor_\delta \in V \quad \lfloor M[M_2] \rfloor_\delta \in V \quad \lfloor M_1 \rfloor_\delta \equiv \lfloor M_2 \rfloor_\delta$$

and we need to prove the following.

$$\lfloor M[M_1] \rfloor_\delta \equiv \lfloor M[M_2] \rfloor_\delta$$

The proof is immediate by applying Lemma 5.

Case MT-DYNINTRO: This case is similar to MT-CONG except that we examine whether δ touches the type being inserted into the context. Specifically, if $\lfloor M_1 \rfloor_\delta$ or $\lfloor M_2 \rfloor_\delta$ yields a type that contains $\star$s, then the implicants of the theorem fail (because implicants require $\lfloor M_1 \rfloor_\delta$ and $\lfloor M_2 \rfloor_\delta$ to be variational type that do not contain $\star$s) and the implication holds vacuously. Otherwise, if neither $\lfloor M_1 \rfloor_\delta$ or $\lfloor M_2 \rfloor_\delta$ contains $\star$s, then $\lfloor M_1 \rfloor_\delta = \lfloor M_2 \rfloor_\delta$ (because M_1 and M_2 differ by that only M_2 replaced some static types with $\star$). This case holds due to the reflexivity (VT-REF) 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*)
$\lfloor M_1 \rfloor_\delta \sim \lfloor M_2 \rfloor_\delta \wedge \lfloor M[M_1] \rfloor_\delta \in G \wedge \lfloor M[M_2] \rfloor_\delta \in G \Rightarrow \lfloor M[M_1] \rfloor_\delta \sim \lfloor M[M_2] \rfloor_\delta$

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_3 if G_2 contains fewer $\star$s than G_3 and they agree on the static parts (Garcia & Cimini, 2015). For example, `Int` is more precise than $\star$ and `Int` but not `Bool`. As another example, `Int` $\to$ `Bool` is more precise than $\star \to$ `Bool` and `Int` $\to \star$ but not $\star \to$ `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 γ or α. 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 a $\star$. G_3 must also be a $\star$, making $G_1 \sim G_3$.
(3) G_2 has the structure $G_{21} \to G_{22}$. If either G_1 or G_3 is a $\star$, then $G_1 \sim G_3$ holds. Otherwise, based on rules C1 or C4 of $\sim$, G_1 has the structure $G_{11} \to G_{12}$ and G_3 has the structure $G_{31} \to G_{32}$. 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 $\approx$, we have $A\langle \texttt{Int}, \texttt{Bool}\rangle \approx A\langle \texttt{Int}, \star\rangle$. Based on that relation, we have $\texttt{Int} \sim \texttt{Int}$ at $A.1$ and $\texttt{Bool} \sim \star$ 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-VTTRANS is also simple since the rule deals with variational types (without $\star$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

$$\lfloor M \rfloor_\delta \in G \qquad \lfloor d\langle M, M\rangle \rfloor_\delta \in G$$

and need to show the following implicand.

$$\lfloor M \rfloor_\delta \sim \lfloor d\langle M, M\rangle \rfloor_\delta$$

From $\lfloor d\langle M, M\rangle \rfloor_\delta \in G$, we know that $d.1 \in \delta$ or $d.2 \in \delta$. Either way, we have $\lfloor d\langle M, M\rangle \rfloor_\delta = \lfloor M \rfloor_\delta$ based on the definition of $\lfloor \cdot \rfloor_\delta$. This case thus holds due to rule C1.

Case MT-DEADELIM: We know the following

$$\lfloor d\langle M_1, M_2\rangle \rfloor_\delta \in G \quad \lfloor d\langle \lfloor M_1 \rfloor_{d.1}, \lfloor M_2 \rfloor_{d.2}\rangle \rfloor_\delta \in G \quad d\langle M_1, M_2\rangle \approx d\langle \lfloor M_1 \rfloor_\delta, \lfloor M_2 \rfloor_\delta\rangle$$

and we need to prove the following relation.

$$\lfloor d\langle M_1, M_2\rangle \rfloor_\delta \sim \lfloor d\langle \lfloor M_1 \rfloor_{d.1}, \lfloor M_2 \rfloor_{d.2}\rangle \rfloor_\delta$$

Both $\lfloor d\langle M_1, M_2\rangle \rfloor_\delta \in G$ and $\lfloor d\langle \lfloor M_1 \rfloor_\delta, \lfloor M_2 \rfloor_\delta\rangle \rfloor_\delta \in G$ 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 δ'. 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:

$$\begin{aligned} \lfloor d\langle M_1, M_2\rangle \rfloor_\delta &= \lfloor \lfloor d\langle M_1, M_2\rangle \rfloor_{d.1} \rfloor_{\delta'} && \delta = \{d.1\} \cup \delta' \\ &= \lfloor \lfloor M_1 \rfloor_{d.1} \rfloor_{\delta'} && \textit{Definition of } \lfloor \cdot \rfloor_\delta \end{aligned}$$

$$\begin{aligned} \lfloor d\langle \lfloor M_1 \rfloor_{d.1}, \lfloor M_2 \rfloor_{d.2} \rangle \rfloor_\delta &= \lfloor \lfloor d\langle \lfloor M_1 \rfloor_{d.1}, \lfloor M_2 \rfloor_{d.2} \rangle \rfloor_{d.1} \rfloor_{\delta'} && \delta = \{d.i\} \cup \delta' \\ &= \lfloor \lfloor \lfloor M_1 \rfloor_{d.1} \rfloor_{d.1} \rfloor_{\delta'} && \textit{Definition of } \lfloor \cdot \rfloor_\delta \\ &= \lfloor \lfloor M_1 \rfloor_{d.1} \rfloor_{\delta'} && \textit{lemma } 3 \end{aligned}$$

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,

$$M_1 \approx M_2[M] \quad \lfloor M_1 \rfloor_\delta \in G \quad \lfloor M_2[M] \rfloor_\delta \in G \quad \lfloor M_1 \rfloor_\delta \sim \lfloor M_2[M] \rfloor_\delta$$

and we need to show that

$$\lfloor M_1 \rfloor_\delta \sim \lfloor M_2[\star] \rfloor_\delta$$

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

□

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 (*$\sqcap$ makes types more precise*)
Let $G_3 = G_1 \sqcap G_2$, 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 $\sqcap$ 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 $G \sim \star$ and need to derive $G \approx \star$. 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 $G \approx [\star]$, which is the same as $G \approx \star$.

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

Case C4: We are given:

$$G_{11} \sim G_{21} \quad G_{12} \sim G_{22} \quad G_{11} \to G_{12} \sim G_{21} \to G_{22}$$

and we need to show:

$$G_{11} \to G_{12} \approx G_{21} \to G_{22}$$

First, let $G_{31} = G_{11} \sqcap G_{21}$ and $G_{32} = G_{12} \sqcap G_{22}$. Through the rule MT-REFL, we have $G_{31} \to G_{32} \approx G_{31} \to G_{32}$. 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 $\star$ to reach G_{11}. Based on this observation, we could repeatedly apply the rule MT-DYNINTRO to $G_{31} \to G_{32} \approx G_{31} \to G_{32}$ to get $G_{31} \to G_{32} \approx G_{11} \to G_{12}$. After that, with MT-SYM, we have $G_{11} \to G_{12} \approx G_{31} \to G_{32}$. We can then repeatedly apply MT-DYNINTRO again to prove $G_{11} \to G_{12} \approx G_{21} \to G_{22}$.

□

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-IDEMP, 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:

$$V_{11} \equiv V_{21} \quad V_{12} \equiv V_{22} \quad V_{11} \to V_{21} \equiv V_{12} \to V_{22}$$

and we have the following by the induction hypotheses

$$V_{11} \approx V_{21} \qquad V_{12} \approx V_{22}$$

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

Case VT-CHOICE: We are given

$$V_{11} \equiv V_{21} \quad V_{12} \equiv V_{22} \quad d\langle V_{11}, V_{21}\rangle \equiv d\langle V_{12}, V_{22}\rangle$$

and have the following by the induction hypotheses:

$$V_{11} \approx V_{21} \qquad V_{12} \approx V_{22}$$

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

□

Before proving the theorem, we present a lemma relating types and the types they produce through selection.

Lemma 11
$\forall \delta . \lfloor M_1 \rfloor_\delta \approx \lfloor M_2 \rfloor_\delta \Rightarrow M_1 \approx M_2$

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 γ and $\forall \delta . \lfloor \gamma \rfloor_\delta = \gamma$.

Case VAR: The proof is direct from the fact that $\lfloor \Gamma \rfloor_\delta$ changes $x \mapsto M$ in the environment to $x \mapsto \lfloor M \rfloor_\delta$. So we can directly use VAR to conclude $\mathit{statifierForDesc}\,(\Omega, \delta); \lfloor \Gamma \rfloor_\delta \vdash_{GC} x : \lfloor M \rfloor_\delta$.

Case ABS: Given the initial typing: $\pi; \Gamma \vdash \lambda x.e : V \to M \,|\, \Omega$ we want to verify that for any δ and some Ω where $\lfloor \pi \rfloor_\delta = \top$, there is a typing:

$$\mathit{statifierForDesc}\,(\Omega, \delta); \lfloor \Gamma \rfloor_\delta \vdash_{GC} \lambda x.e : \lfloor V \to M \rfloor_\delta$$

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:

$$\mathit{statifierForDesc}\,(\Omega, \delta); \lfloor \Gamma \rfloor_\delta, x \mapsto \lfloor V \rfloor_\delta \vdash_{GC} e : \lfloor M \rfloor_\delta$$

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

$$\mathit{statifierForDesc}\,(\Omega, \delta); \lfloor \Gamma \rfloor_\delta \vdash_{GC} \lambda x.e : \lfloor V \rfloor_\delta \to \lfloor M \rfloor_\delta$$

where $\lfloor V \rfloor_\delta \to \lfloor M \rfloor_\delta = \lfloor V \to M \rfloor_\delta$.

Case ABSDYN: Given the initial typing:

$$\pi; \Gamma \vdash \lambda x : \star . e : d\langle \star, V \rangle \to M \,|\, \Omega \cup \{x \mapsto V\}$$

we want to show that for any δ and some ω where $\lfloor \pi \rfloor_\delta = \top$ we have the following:

$$\omega; \lfloor \Gamma \rfloor_\delta \vdash_{GC} \lambda x : \star . e : \omega(x) \to \lfloor M \rfloor_\delta$$

When constructing the inital typing we had the following premise:

$$\pi; \Gamma, x \mapsto d\langle \star, V \rangle \vdash e : M \,|\, \Omega$$

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

$$\mathit{statifierForDesc}\,(\Omega, \delta); \lfloor \Gamma \rfloor_\delta, x \mapsto \lfloor d\langle \star, V \rangle \rfloor_\delta \vdash_{GC} e : \lfloor M \rfloor_\delta$$

Since we do not know whether Ω 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 $\lfloor d\langle\star,V\rangle\rfloor_\delta$ must produce either $\star$ (if $d.1$ in δ) or some static type, T, where $\lfloor V\rfloor_\delta$=T. Consequently, we can infer that *statifierForDesc* $(\Omega',\delta)(x)$ either produces $\star$ when $d.1 \in \delta$ or T when $d.2 \in \delta$. Let ω=*statifierForDesc* (Ω',δ). We can now derive:

$$\omega;\lfloor\Gamma\rfloor_\delta, x \mapsto \omega(x) \vdash_{GC} e : \lfloor M\rfloor_\delta$$

Now we can use AbsDyn to conclude:

$$\omega;\lfloor\Gamma\rfloor_\delta \vdash_{GC} \lambda x:\star.e : \omega(x) \to \lfloor M\rfloor_\delta$$

where ω(x)=$\lfloor d\langle\star,V\rangle\rfloor_\delta$.

Case App: We are given the initial typing:

$$\pi;\Gamma \vdash e_1\ e_2 : cod_\pi(M_1)\,|\,\Omega$$

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

$$statifierForDesc\,(\Omega,\delta);\lfloor\Gamma\rfloor_\delta \vdash_{GC} e_1\ e_2 : cod(\lfloor M_1\rfloor_\delta)$$

for any δ and some Ω such that $\lfloor\pi\rfloor_\delta$=$\top$. In constructing the initial typing we had the following premises:

$$\pi;\Gamma \vdash e_1 : M_1\,|\,\Omega_1 \quad \pi;\Gamma \vdash e_2 : M_2\,|\,\Omega_2 \quad dom_\pi(M_1) \approx_\pi M_2$$

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

$$statifierForDesc\,(\Omega_1,\delta);\lfloor\Gamma\rfloor_\delta \vdash_{GC} e_1 : \lfloor M_1\rfloor_\delta \qquad statifierForDesc\,(\Omega_2,\delta);\lfloor\Gamma\rfloor_\delta \vdash_{GC} e_2 : \lfloor M_2\rfloor_\delta$$

Let ω = *statifierForDesc* (Ω_1,δ) $\cup$ *statifierForDesc* (Ω_2,δ), the following two typing relations are satisfied because we can rename parameter names so that *statifierForDesc* (Ω_1,δ) (*statifierForDesc* (Ω_2,δ)) be a subset of $\omega$and enlarge *statifierForDesc* (Ω_1,δ) (*statifierForDesc* (Ω_2,δ)) does not change the typing result of the first (second) typing relation above.

$$\omega;\lfloor\Gamma\rfloor_\delta \vdash_{GC} e_1 : \lfloor M_1\rfloor_\delta \qquad \omega;\lfloor\Gamma\rfloor_\delta \vdash_{GC} e_2 : \lfloor M_2\rfloor_\delta$$

Given that $\pi;\Gamma \vdash e_1\ e_2 : cod_\pi(M_1)\,|\,\Omega$, we have the following result

$$\begin{aligned} dom_\pi(M_1) \approx_\pi M_2 &\Rightarrow dom_\top(\lfloor M_1\rfloor_\delta) \approx_\top \lfloor M_2\rfloor_\delta && \lfloor\pi\rfloor_\delta = \top \\ &\Rightarrow dom(\lfloor M_1\rfloor_\delta) \sim \lfloor M_2\rfloor_\delta && \textit{Theorem 2} \end{aligned}$$

We can now use App to conclude:

$$\omega;\lfloor\Gamma\rfloor_\delta \vdash_{GC} e_1\ e_2 : cod(\lfloor M_1\rfloor_\delta)$$

Case If: The proof for this case is similar to that for the App case and is omitted here.

Case Weaken: This rule can only modify selections on M where a decision δ' yields $\lfloor\pi\rfloor_{\delta'}$=$\bot$. Since the theorem requires $\lfloor\pi\rfloor_\delta = \top$, 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 CON, VAR, and ABS are straightforward since they do not introduce variations and have at most one subexpression.

Case ABSDYN: We have the initial typing:

$$\omega;\Gamma \vdash_{GC} e : \omega(x) \to G$$

and we need to derive the following:

$$\pi;\Gamma \vdash \lambda x.e : M' \,|\, \Omega$$

where $\lfloor M' \rfloor_\delta = \omega(x) \to G$ and there is some Ω such that $\mathit{statifierForDesc}(\Omega,\delta) = \omega$. We have the following premise when constructing the inital typing:

$$\omega;\Gamma, x \mapsto \omega(x) \vdash_{GC} e : G$$

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:

$$\pi;\Gamma, x \mapsto T \vdash e : M \,|\, \Omega \quad \lfloor M \rfloor_\delta = G \quad \mathit{statifierForDesc}(\Omega,\delta) = \omega$$

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

$$\pi;\Gamma, x \mapsto \star \vdash e : M' \,|\, \Omega$$

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

$$\pi;\Gamma, x \mapsto d\langle \star, T\rangle \vdash e : d\langle M', M\rangle \,|\, \Omega$$

Since the function was well typed in ITGL, we can use $\top$ when typing the variational version. Then we can use ABSDYN to derive:

$$\top;\Gamma \vdash \lambda x.e : d\langle \star, T\rangle \to d\langle M', M\rangle \,|\, \Omega \cup \{x \mapsto T\}$$

Let $\Omega' = \Omega \cup \{x \mapsto T\}$. We next show that $\mathit{statifierForDesc}(\Omega',\delta)$ and $\lfloor d\langle \star, T\rangle \to d\langle M', M\rangle \rfloor_\delta = T \to G$. 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 $\mathit{statifierForDesc}(\Omega,\delta) = \omega$, and thus $x \mapsto V \in \Omega$, where $\mathit{statifierForDesc}(\Omega,\delta)(x) = \lfloor V \rfloor_\delta = T$. Consequently, $\mathit{statifierForDesc}(\Omega,\delta) = \mathit{statifierForDesc}(\Omega',\delta) = \omega$. Moreover, we know that:

$$\begin{aligned}
\lfloor d\langle \star, T\rangle \to d\langle M', M\rangle \rfloor_\delta &= \lfloor d\langle \star, T\rangle \rfloor_\delta \to \lfloor d\langle M', M\rangle \rfloor_\delta \\
&= T \to \lfloor M \rfloor_\delta && d.2 \in \delta \\
&= T \to G && I.H
\end{aligned}$$

Case APP: We have the initial typing:

$$\omega_1 \cup \omega_2;\Gamma \vdash_{GC} e_1\ e_2 : \mathit{cod}(G_1)$$

We need to derive:

$$\pi;\Gamma \vdash e_1\ e_2 : cod_\pi(M_1)\,|\,\Omega$$

with $\lfloor cod_\pi(M_1)\rfloor_\delta = cod(G_1)$ and there is some Ω such that $statifierForDesc(\Omega,\delta) = \omega$. From the derivation of the initial typing we have the following premises:

$$\omega_1;\Gamma \vdash_{GC} e_1 : G_1 \quad \omega_2;\Gamma \vdash_{GC} e_2 : G_2 \quad dom(G_1) \sim G_2$$

By the induction hypothesis and these premises, we have:

$$\begin{array}{llll} \pi_1;\Gamma \vdash e_1 : M_1\,|\,\Omega_1 & \lfloor \pi_1 \rfloor_{\delta_1} = \top & \lfloor M_1 \rfloor_{\delta_1} = G_1 & statifierForDesc(\Omega_1,\delta_1){=}\omega_1 \\ \pi_2;\Gamma \vdash e_2 : M_2\,|\,\Omega_2 & \lfloor \pi_1 \rfloor_{\delta_1} = \top & \lfloor M_2 \rfloor_{\delta_2} = G_2 & statifierForDesc(\Omega_2,\delta_2){=}\omega_2 \end{array}$$

Let $\delta' = \delta_1 \cup \delta_2$, $\pi' = \pi_1 \sqcap \pi_2$, and π be a pattern such that $\lfloor \pi \rfloor_{\delta'} = \lfloor \pi' \rfloor_{\delta'}$ and $\forall \delta.\delta \neq \delta' \Rightarrow \lfloor \pi \rfloor_\delta = \bot$. As a result, $\lfloor \pi \rfloor_{\delta'} = \lfloor \pi' \rfloor_{\delta'} = \lfloor \pi_1 \sqcap \pi_2 \rfloor_{\delta_1 \cup \delta_2} = \lfloor \pi_1 \rfloor_{\delta_1 \cup \delta_2} \sqcap \lfloor \pi_2 \rfloor_{\delta_1 \cup \delta_2} = \lfloor \top \rfloor_{\delta_2} \sqcap \lfloor \top \rfloor_{\delta_1} = \top \sqcap \top = \top$.

Based on the construction of π, $\pi \leq \pi' \leq \pi_1$. Thus, we have $\pi;\Gamma \vdash e_1 : M_1\,|\,\Omega_1$ based on Weaken. Similarly, we have $\pi;\Gamma \vdash e_2 : M_2\,|\,\Omega_2$. Moreover, based on $\lfloor M_1 \rfloor_{\delta_1} = G_1$, we have $\lfloor M_1 \rfloor_\delta = G_1$ since $\delta_1 \subseteq \delta$. Similarly, we have $\lfloor M_2 \rfloor_\delta = G_2$. From $dom(G_1) \sim G_2$ and the construction of π, we have $dom_\pi(M_1) \approx_\pi M_2$ based on Theorem 3. Therefore, we have $\pi;\Gamma \vdash e_1\ e_2 : cod_\pi(M_1)\,|\,\Omega$, where $\Omega = \Omega_1 \cup \Omega_2$.

As Ω_1 and Ω_2 are used to type different subexpressions, their domains are disjoint, and so do ω_1 and ω_2. As a result $statifierForDesc(\Omega,\delta') = statifierForDesc(\Omega_1,\delta_1) \cup statifierForDesc(\Omega_2,\delta_2) = \omega$. Since $\lfloor M_1 \rfloor_{\delta_1} = G_1$, we have $\lfloor cod_\pi(M_1) \rfloor_\delta = cod(G_1)$. This completes the proof for this case.

The case for If can be proved similarly to the 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.

$$\begin{array}{ll} \top \sqcup \pi = \top & d\langle \pi_1, \pi_2 \rangle \sqcup d\langle \pi_3, \pi_4 \rangle = d\langle \pi_1 \sqcup \pi_3, \pi_2 \sqcup \pi_4 \rangle \\ \bot \sqcup \pi = \pi & d\langle \pi_1, \pi_2 \rangle \sqcup \pi = d\langle \pi_1 \sqcup \pi, \pi_2 \sqcup \pi \rangle \end{array}$$

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

1. $\pi_1 \leq \pi \wedge \pi_2 \leq \pi \Rightarrow \pi_1 \sqcup \pi_2 \leq \pi$
2. $M \approx_{\pi_1} M_1 \wedge M \approx_{\pi_2} M_1 \Rightarrow M \approx_{\pi_1 \sqcup \pi_2} M_1$
3. $M\ op_{\pi_1}\ M_1 \wedge M\ op_{\pi_2}\ M_1 \Rightarrow M\ op_{\pi_1 \sqcup \pi_2}\ M_1$
4. $op_{\pi_1}(M) \wedge op_{\pi_2}(M) \Rightarrow op_{\pi_1 \sqcup \pi_2}(M)$

The proofs of these properties follow directly from induction over π, 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:

$$\pi_1;\Gamma \vdash e_1\ e_2 : cod_{\pi_1}(M_1)\,|\,\Omega \qquad \pi_2;\Gamma \vdash e_1\ e_2 : cod_{\pi_2}(M_1)\,|\,\Omega$$

and need to prove the following relation

$$\pi_3;\Gamma \vdash e_1\ e_2 : cod_{\pi_3}(M_1)\,|\,\Omega$$

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

$$\begin{array}{ll} \pi_1;\Gamma \vdash e_1 : M_1\,|\,\Omega & \pi_1;\Gamma \vdash e_2 : M_2\,|\,\Omega \\ \pi_2;\Gamma \vdash e_1 : M_1\,|\,\Omega & \pi_2;\Gamma \vdash e_2 : M_2\,|\,\Omega \end{array}$$

By the induction hypothesis and these premises, we have:

$$\begin{array}{cc} \pi_3';\Gamma \vdash e_1 : M_1\,|\,\Omega & \pi_3';\Gamma \vdash e_2 : M_2\,|\,\Omega \\ \pi_1 \leq \pi_3' & \pi_2 \leq \pi_3' \end{array}$$

We take $\pi_3 = \pi_1 \sqcup \pi_2$ and we must now show that π_3 can be used in the typing of the implicand. To type the implicants with π_3 we know that $dom_{\pi_1}(M_1)$ and $dom_{\pi_2}(M_1)$ must be defined. Based on Lemma 12 property 3 and the definition of π_3, we have:

$$dom_{\pi_1}(M_1) \wedge dom_{\pi_2}(M_1) \Rightarrow dom_{\pi_3}(M_1)$$

Similarly, we can see that that $dom_{\pi_3}(M_1) \approx_{\pi_3} M_2$ 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 π_3 to derive $\pi_3;\Gamma \vdash e_1 : M_1\,|\,\Omega$ and $\pi_3;\Gamma \vdash e_2 : M_2\,|\,\Omega$. Pairing those typings with $dom_{\pi_3}(M_1) \approx_{\pi_3} M_2$, we can use App to conclude:

$$\pi_3;\Gamma \vdash e_1\ e_2 : cod_{\pi_3}(M_1)\,|\,\Omega$$

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 $cod_\pi(M_1) \preceq cod_\pi(M_2)$.

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 γ_1 and γ_2 represent two different static constant types. From $\sqcap^\alpha$, we derive $\sqcap^\alpha_\pi$ as we did for deriving $\sqcap_\pi$ from $\sqcap$ and as for deriving dom_π from dom (Section 4.3).

Lemma 14 (*Typing under different assumptions*)
For any e and Γ, if $\pi;\Gamma,x\mapsto d\langle\star,V_1\rangle\vdash e:M_1\,|\,\Omega_1$ and $\pi;\Gamma,x\mapsto d\langle\star,V_2\rangle\vdash e:M_2\,|\,\Omega_2$, then $\pi;\Gamma,x\mapsto d\langle\star,V_1\sqcap^{\alpha}_{\pi}V_2\rangle\vdash e:M_3\,|\,\Omega_3$, $M_1\preceq M_3$, $M_2\preceq M_3$, $\Omega_1\preceq\Omega_3$, and $\Omega_2\preceq\Omega_3$.

In the Lemma, we write $\omega_1\preceq\omega_2$ if ω_1 and ω_2 share the same domain and if for any x in the domain $\omega_1(x)\preceq\omega_2(x)$.

The proof is an induction over the typing rules in Figure 10. The case Con is immediate. For the case 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 $V_1\preceq V_1\sqcap^{\alpha}_{\pi}V_2$ and $V_2\preceq V_1\sqcap^{\alpha}_{\pi}V_2$. The proof for cases Abs and AbsDyn are based on simple inductions. The proof for App and If is similar to the proof of these cases for Lemma 2 and is omitted here. The proof for Weaken is based on a simple induction.

Proof of Lemma 2
The proof follows by induction over the rules in Figure 10. Cases Con and Var are straightforward and omitted for brevity. Case Abs is also omitted since it is similar to AbsDyn, covered below.

Case AbsDyn: We are given with the following:

$$\pi;\Gamma\vdash\lambda x:\star.e:d\langle\star,V_1\rangle\to M_1\,|\,\Omega_1\cup\{x\mapsto V_1\}\qquad \pi;\Gamma\vdash\lambda x:\star.e:d\langle\star,V_2\rangle\to M_2\,|\,\Omega_2\cup\{x\mapsto V_2\}$$

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

$$\pi;\Gamma\vdash\lambda x:\star.e:M\,|\,\Omega$$

where $d\langle\star,V_1\rangle\to M_1\preceq M$ and $d\langle\star,V_2\rangle\to M_2\preceq M$ for some M and $\Omega_1\cup\{x\mapsto V_1\}\preceq\Omega$ and $\Omega_2\cup\{x\mapsto V_2\}\preceq\Omega$ for some Ω.
From the construction of the implicants, we know the following premises:

$$\pi;\Gamma,x\mapsto d\langle\star,V_1\rangle\vdash e:M_1\,|\,\Omega_1\qquad \pi;\Gamma,x\mapsto d\langle\star,V_2\rangle\vdash e:M_2\,|\,\Omega_2$$

Based on Lemma 14, let $V_3=V_1\sqcap^{\alpha}_{\pi}V_2$, we can construct the typing:

$$\pi;\Gamma,x\mapsto d\langle\star,V_3\rangle\vdash e:M_3\,|\,\Omega_3$$

with $d\langle\star,V_1\rangle\preceq d\langle\star,V_3\rangle$, $d\langle\star,V_2\rangle\preceq d\langle\star,V_3\rangle$, $M_1\preceq M_3$, $M_2\preceq M_3$, $\Omega_1\preceq\Omega_3$, and $\Omega_2\preceq\Omega_3$. With AbsDyn, we can derive the following typing relation:

$$\pi;\Gamma\vdash\lambda x.e:d\langle\star,V_3\rangle\to M_3\,|\,\Omega_3\cup\{x\mapsto V_3\}$$

Moreover, $d\langle\star,V_1\rangle\to M_1\preceq d\langle\star,V_3\rangle\to M_3$ and $d\langle\star,V_2\rangle\to M_2\preceq d\langle\star,V_3\rangle\to M_3$, 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 $\mathit{statifierForDesc}\,(\Omega_1',\delta)\preceq\mathit{statifierForDesc}\,(\Omega_3',\delta)$ $\mathit{statifierForDesc}\,(\Omega_2',\delta)\preceq\mathit{statifierForDesc}\,(\Omega_3',\delta)$.
Case App: Given the following judgments:

$$\pi;\Gamma\vdash e_1\ e_2:cod_{\pi}(M_{11})\,|\,\Omega_1\qquad \pi;\Gamma\vdash e_1\ e_2:cod_{\pi}(M_{21})\,|\,\Omega_2$$

we want to prove the following typing derivation:

$$\pi;\Gamma\vdash e_1\ e_2:cod_{\pi}(M_{31})\,|\,\Omega_3$$

where Ω_3 and $cod_\pi(M_{31})$ are the best variational statifier and type in the three derivations. In typing the implicants, we had the following premises:

$$\begin{array}{ccc} \pi;\Gamma \vdash e_1 : M_{11} \,|\, \Omega_{11} & \pi;\Gamma \vdash e_2 : M_{12} \,|\, \Omega_{12} & dom_\pi(M_{11}) \approx^? M_{12} \\ \pi;\Gamma \vdash e_1 : M_{21} \,|\, \Omega_{21} & \pi;\Gamma \vdash e_2 : M_{22} \,|\, \Omega_{22} & dom_\pi(M_{21}) \approx^? M_{22} \end{array}$$

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:

$$\begin{array}{ccc} \pi;\Gamma \vdash e_1 : M_{31} \,|\, \Omega_{31} & \pi;\Gamma \vdash e_2 : M_{32} \,|\, \Omega_{32} & dom_\pi(M_{31}) \approx^? M_{32} \\ \lfloor M_{11} \rfloor_\delta \preceq \lfloor M_{31} \rfloor_\delta & \multicolumn{2}{c}{statifierForDesc\,(\Omega_{11},\delta) \preceq statifierForDesc\,(\Omega_{31},\delta)} \\ \lfloor M_{12} \rfloor_\delta \preceq \lfloor M_{32} \rfloor_\delta & \multicolumn{2}{c}{statifierForDesc\,(\Omega_{12},\delta) \preceq statifierForDesc\,(\Omega_{32},\delta)} \\ \lfloor M_{21} \rfloor_\delta \preceq \lfloor M_{31} \rfloor_\delta & \multicolumn{2}{c}{statifierForDesc\,(\Omega_{21},\delta) \preceq statifierForDesc\,(\Omega_{31},\delta)} \\ \lfloor M_{22} \rfloor_\delta \preceq \lfloor M_{32} \rfloor_\delta & \multicolumn{2}{c}{statifierForDesc\,(\Omega_{12},\delta) \preceq statifierForDesc\,(\Omega_{32},\delta)} \end{array}$$

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

Now note that $\lfloor M_{11} \rfloor_\delta \preceq \lfloor M_{31} \rfloor_\delta$ and $\lfloor M_{12} \rfloor_\delta \preceq \lfloor M_{32} \rfloor_\delta$ imply $\lfloor cod_\pi(M_{11}) \rfloor_\delta \preceq \lfloor cod_\pi(M_{31}) \rfloor_\delta$ and $\lfloor cod_\pi(M_{21}) \rfloor_\delta \preceq \lfloor cod_\pi(M_{31}) \rfloor_\delta$ 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.

$$\pi;\Gamma \vdash e_1\ e_2 : cod_\pi(M_{31}) \,|\, \Omega_3$$

Case IF: 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:

$$\pi_3;\Gamma \vdash e : M_3 \,|\, \Omega_3$$

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

$$\begin{array}{cc} \pi;\Gamma \vdash e : M_1 \,|\, \Omega_1 & \pi;\Gamma \vdash e : M_2 \,|\, \Omega_2 \\ \pi_1 \leq \pi & \pi_2 \leq \pi \\ M_1 =_{\pi_1} M_1' & M_2 =_{\pi_2} M_2' \end{array}$$

By the induction hypothesis and the premises, we have:

$$\begin{array}{cc} \multicolumn{2}{c}{\pi;\Gamma \vdash e : M_3 \,|\, \Omega_3} \\ \lfloor M_1 \rfloor_\delta \preceq \lfloor M_3 \rfloor_\delta & statifierForDesc\,(\Omega_1,\delta) \preceq statifierForDesc\,(\Omega_3,\delta) \\ \lfloor M_2 \rfloor_\delta \preceq \lfloor M_3 \rfloor_\delta & statifierForDesc\,(\Omega_2,\delta) \preceq statifierForDesc\,(\Omega_3,\delta) \end{array}$$

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

$$\pi_3;\Gamma \vdash e : M_3 \,|\, \Omega_3$$

From here, applying our induction hypothesis to the premises tell us that $statifierForDesc\,(\Omega_1,\delta) \preceq statifierForDesc\,(\Omega_3,\delta)$ and $statifierForDesc\,(\Omega_2,\delta) \preceq$

statifierForDesc (Ω_3, δ), completing the part of the proof involving statifiers. From here we just need to show $\lfloor M_1' \rfloor_\delta \preceq \lfloor M_3 \rfloor_\delta$ and $\lfloor M_2' \rfloor_\delta \preceq \lfloor M_3 \rfloor_\delta$ 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 $\lfloor \pi \rfloor_\delta = \top$:

$$\lfloor M_1 \rfloor_\delta = \lfloor M_1' \rfloor_\delta$$

From this and by our induction hypothesis we can conclude:

$$\lfloor M_1' \rfloor_\delta \preceq \lfloor M_3 \rfloor_\delta$$

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.

□

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 *domCst* $(M_a, M_b) \hookrightarrow C$ and (θ,π) is sound for C, then $dom_\pi(\theta(M_a)) \approx_\pi \theta(M_b)$.
- If *codCst* $(M_a) \hookrightarrow (M_b, C)$ and (θ,π) is sound for C, then $cod_\pi(\theta(M_a)) =_\pi \theta(M_b)$.
- If $M_a \sqcap M_b \hookrightarrow (M_c, C)$ and (θ,π) is sound for C, then $\theta(M_a) \sqcap_\pi \theta(M_b) \approx_\pi \theta(M_c)$.

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 *domCst* and proving the lemma holds.

Case 1 In this case, we have $M_a = \star$ and $M_b = M$. The generated constraint is ε. The sound solution for this constraint is $(\emptyset, \top)$. We know that $dom_\pi(\theta(\star))$ is $\star$, which is compatible with $\theta(M)$.

Case 2 In this case, $M_a = \alpha$, $M_b = M$, and the generated constraint is $\alpha \approx^? M \to \kappa_2$. Since (θ,π) is sound for this constraint, we have $\theta(\alpha) \approx_\pi \theta(M \to \kappa_2) = \theta(M) \to \theta(\kappa_2)$. It is immediate that $dom_\pi(\theta(M_a)) = \theta(M_b)$.

Case 3 In this case, $M_a = M_{11} \to M_{12}$ and $M_b = M$. The constraint is $M_{11} \approx^? M$. By definition, (θ,π) is sound for this constraint means that $\theta(M_{11}) \approx_\pi \theta(M)$. Since $dom_\pi(\theta(M_a))$ $= dom_\pi(\theta(M_{11} \to M_{12})) = \theta(M_{11})$, we have $dom_\pi(\theta(M_a)) \approx_\pi \theta(M_b)$.

Case 4 In this case, $M_a = d\langle M_1, M_2 \rangle$, $M_b = M$, and the constraint is $d\langle domCst(M_1, M), domCst(M_2, M) \rangle$. Assume (θ,π) is a sound solution for the constraint $d\langle domCst(M_1, M), domCst(M_2, M) \rangle$. It will also be sound for *domCst* (M_1, M) and *domCst* (M_2, M). As a result, we have $dom_\pi(\theta(M_1)) \approx_\pi \theta(M)$

and $dom_\pi(\theta(M_b)) \approx_\pi \theta(M)$. Now

$$\begin{aligned}
dom_\pi(\theta(M_a)) &= dom_\pi(\theta(d\langle M_1, M_2\rangle)) \\
&= dom_\pi(d\langle \theta(M_1), \theta(M_2)\rangle) && \text{based on the definition of substitution} \\
&= d\langle dom_\pi(\theta(M_1)), dom_\pi(\theta(M_2))\rangle && \text{based on the definition of } dom \text{(Figure 10)} \\
&\approx_\pi d\langle \theta(M), \theta(M_2)\rangle && \text{see above} \\
&= \theta(M) && \text{due to choice idempotency} \\
&= \theta(M_b)
\end{aligned}$$

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 $dom_\bot(\theta(M_a)) \approx_\bot \theta(M_b)$ holds.

□

Proof of Theorem 11

The proof proceeds by induction over the constraint generation rules in Figure 11. Cases VARC and 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.

$$\begin{array}{ccc}
\Gamma \vdash_C e_1 : M_1 \mid C_1 & \Gamma \vdash_C e_2 : M_2 \mid C_2 & \\
domCst\,(M_1, M_2) \hookrightarrow C_4 & codCst\,(M_1) \hookrightarrow (M_3, C_3) & C = C_1 \wedge C_2 \wedge C_3 \wedge C_4
\end{array}$$

we want to produce the typing derivation:

$$\pi; \theta(\Gamma) \vdash e_1\ e_2 : \theta(M_3) \mid \Omega$$

Since (θ, π) 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

$$\pi; \theta(\Gamma) \vdash e_1 : \theta(M_1) \mid \Omega_1 \qquad \pi; \theta(\Gamma) \vdash e_2 : \theta(M_2) \mid \Omega_2$$

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 $\pi;\theta(\Gamma) \vdash e_1\ e_2 : \theta(cod_\pi(M_1))\,|\,\Omega$. Based on the fourth premise and Lemma 15 we know that $dom_\pi(M_1) \approx_\pi M_2$. $\theta(cod_\pi(M_1)) = \theta(M_3)$, which means we have $\pi;\theta(\Gamma) \vdash e_1\ e_2 : \theta(M_3)\,|\,\Omega$.

Case IFC: 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 $dom_\pi(\theta(M_a)) \approx_\pi \theta(M_b)$, $domCst(M_a,M_b) \hookrightarrow C$, and (θ_1,π_1) is the sound and most general solution for C, then $\pi \leq \pi_1$ and $\theta_1 \sqsubseteq \theta$.
- If $cod_\pi(\theta(M_a)) =_\pi \theta(M_b)$, $codCst(M_a) \hookrightarrow (M_b,C)$, and (θ_1,π_1) be the sound and most general solution for C, then $\pi \leq \pi_1$ and $\theta_1 \sqsubseteq \theta$.
- If $\theta(M_a) \sqcap_\pi \theta(M_b) \approx_\pi \theta(M_c)$, $M_a \sqcap M_b \hookrightarrow (M_c,C)$, and (θ,π) 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 *domCst*, the generated constraint is $\alpha \approx^? M_b \rightarrow \kappa_2$. As (θ_1,π_1) is sound and most general for this constraints, we have $\theta_1(\alpha) \approx_{\pi_1} \theta_1(M_b \rightarrow \kappa_2)$. Since κ_2 is a fresh unification type variable, $(\theta_1 \cup \{\kappa_2 \mapsto M_2\},\pi_1)$ is sound and most general for the problem $\alpha \approx^? M_b \rightarrow M_2$. As (θ,π) is also sound for this problem, $(\theta_1 \cup \{\kappa_2 \mapsto M_2\},\pi_1)$ is more general than (θ,π). Consequently, (θ_1,π_1) is more general than (θ,π).

In the second subcase, $M_a = M_1' \rightarrow M_2'$. Based on the definition of *domCst*, the generated constraint is $M_1' \approx^? M_b$. Since (θ_1,π_1) is most sound and general, we have $\theta_1(M_1') \approx_{\pi_1} \theta_1(M_b)$. Moreover, based on the condition of the lemma, we have $dom_\pi(\theta(M_a)) \approx_\pi \theta(M_b)$, meaning that $\theta(M_1') \approx_\pi \theta(M_b)$. Overall, both (θ_1,π_1) and (θ,π) are solutions for the same constraint $M_1' \approx^? M_b$ and (θ_1,π_1) is most general. (θ_1,π_1) is more general than (θ,π).

Case 2 In this case $\theta(M_a) = \star$. Since θ maps type variables to static types only, $M_a = \star$. Based on the definition of *domCst*, the generated constraint is ε. The most general solution for it is $(\emptyset,\top)$, which is more general than (θ,π).

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

$$dom_{\lfloor\pi\rfloor_{d.1}}(\lfloor\theta\rfloor_{d.1}(M_1')) \approx_{\lfloor\pi\rfloor_{d.1}} \lfloor\theta\rfloor_{d.1}(M_b) \qquad \text{(A 1)}$$

$$dom_{\lfloor\pi\rfloor_{d.2}}(\lfloor\theta\rfloor_{d.2}(M_2')) \approx_{\lfloor\pi\rfloor_{d.2}} \lfloor\theta\rfloor_{d.2}(M_b) \qquad \text{(A 2)}$$

The constraint generated by *domCst* is $d\langle M_1', M_2'\rangle \approx^? M_b$, which equals $d\langle M_1' \approx^? M_b, M_2' \approx^? M_b\rangle$ based on the definition of *domCst*. Let (θ_{1l}, π_{1l}) be the sound and most general solution for $M_1' \approx^? M_b$. Based on the equation (1) above and the induction hypothesis, (θ_{1l}, π_{1l}) is more general than $(\lfloor\theta\rfloor_{d.1}, \lfloor\pi\rfloor_{d.1})$. Similarly, let (θ_{1r}, π_{1r}) be the sound and most general solution for $M_2' \approx^? M_b$. Based on equation (2) above and the induction hypothesis, (θ_{1r}, π_{1r}) is more general than $(\lfloor\theta\rfloor_{d.2}, \lfloor\pi\rfloor_{d.2})$. As a result, $(d\langle\theta_{1l}, \theta_{1r}\rangle, d\langle\pi_{1l}, \pi_{1r}\rangle)$ is sound and most general for $d\langle M_1', M_2'\rangle \approx^? M_b$, which is more general than $(d\langle\lfloor\theta\rfloor_{d.1}, \lfloor\theta\rfloor_{d.2}\rangle, d\langle\lfloor\pi\rfloor_{d.1}, \lfloor\pi\rfloor_{d.2}\rangle)$ = (θ, π).

□

In proving Theorem 12 below, we need to combine several patterns into one. Specifically, given two patterns π_1 and π_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).

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

$$\bot \sqcap \pi = \bot \qquad d\langle\pi_1, \pi_2\rangle \sqcap \pi = d\langle\pi_1 \sqcap \pi, \pi_2 \sqcap \pi\rangle$$

Intuitively, $\pi_1 \sqcap \pi_2$ contains $\top$s at where both π_1 and π_2 contain $\top$s. If either π_1 or π_2 or both contain $\bot$ at a variant, then $\pi_1 \sqcap \pi_2$ also contains a $\bot$ at that variant. For example, $\top \sqcap \top$ is $\top$, $\top \sqcap A\langle\bot, \top\rangle$ is $A\langle\bot, \top\rangle$, and $A\langle\bot, \top\rangle \sqcap A\langle\top, \bot\rangle$ is $A\langle\bot, \bot\rangle$, which is the same as $\bot$.

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

Lemma 17 ($\sqcap$ *preserves the less-defined relation*)
If $\pi \leq \pi_1$ and $\pi \leq \pi_2$, then $\pi \leq \pi_1 \sqcap \pi_2$.

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

Proof of Theorem 12
This theorem is proven by structural induction over the rules in Figure 10, with help from Lemma 16. Cases VAR and CON are straightforward, so their presentation is omitted.

Case ABS: We are given the following:

$$\pi; \theta(\Gamma) \vdash \lambda x.e : V \to M \mid \Omega, \text{ which has the premise: } \pi; \theta(\Gamma), x \mapsto V \vdash e : M \mid \Omega$$

From the premise, we know that there is some type variable V' such that $V \preceq V'$ and $\theta(V') = V$. We thus have $\pi; \theta(\Gamma, x \mapsto V') \vdash e : M \mid \Omega$. Based on the induction hypothesis, we have

$$\Gamma, x \mapsto V' \vdash_C e : M_1 \mid C \quad \forall\delta \lfloor\pi\rfloor_\delta = \top . \lfloor M\rfloor_\delta \preceq \lfloor\theta_1(M_1)\rfloor_\delta \quad \pi \leq \pi_1 \quad \theta = \theta' \circ \theta_1$$

where (θ_1,π_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 (θ,π) still hold. Moreover, since θ_1 is more general than θ, $V = \theta(V') \preceq \theta_1(V')$. Therefore, $\lfloor V \to M \rfloor_\delta \preceq \lfloor \theta_1(V_1 \to M_1) \rfloor_\delta$ based on the induction hypothesis above.

Case AbsDyn: This case proceeds similarly to Abs.

Case App: We are given the following premises:

$$\pi;\theta(\Gamma) \vdash e_1 : M_1' \,|\, \Omega_1 \quad \pi;\theta(\Gamma) \vdash e_2 : M_2' \,|\, \Omega_2 \quad dom_\pi(M_1') \approx_\pi M_2' \quad M_3' = cod_\pi(M_1')$$

We want to derive:

$$\Gamma \vdash_C e_1\ e_2 : M_3 \mid C$$

such that if (θ_1,π_1) is the solution for C, then $\pi \leq \pi_1$, $\theta_1 \sqsubseteq \theta$, and $M_3' \preceq \theta_1(M_3)$.

We have the following induction hypotheses:

$$\begin{array}{llll} \Gamma \vdash_C e_1 : M_1 \mid C_1 & M_1' \preceq \theta_{11}(M_1) & \pi' \leq \pi_{11} & \theta = \theta_{11}' \circ \theta_{11} \\ \Gamma \vdash_C e_2 : M_2 \mid C_2 & M_2' \preceq \theta_{12}(M_2) & \pi' \leq \pi_{12} & \theta = \theta_{12}' \circ \theta_{12} \end{array}$$

where (θ_{11},π_{11}) solves C_1 and (θ_{12},π_{12}) solves C_2. From $dom_\pi(M_1') \approx_\pi M_2'$, we have $dom_\pi(\theta(M_1)) \approx_\pi \theta(M_2)$. Let $domCst(M_1,M_2) \hookrightarrow C_3$ and (θ_{13},π_{13}) be the solution for C_3, then, based on Lemma 16, we have $\theta = \theta_{13}' \circ \theta_{13}$ and $\pi \leq \pi_{13}$ for some θ_{13}'. Similarly, from $M_3' \approx_\pi cod_\pi(M_1')$ we have $\theta(M_3) \approx_\pi \theta(cod_\pi(M_1))$. Let $codCst(M_1) \hookrightarrow C_4$ and (θ_{14},π_{14}) be the solution for C_4, then based on Lemma 16, we have $\theta = \theta_{14}' \circ \theta_{14}$ and $\pi \leq \pi_{14}$ for some θ_{14}'.

We can now use AppC to derive the following relation

$$\Gamma \vdash_C e_1\ e_2 : M_3 \mid C_1 \wedge C_2 \wedge C_3 \wedge C_4$$

Moreover, for each C_i, we have (θ_{1i},π_{1i}) that is more general than (θ,π). We next need to prove that we can combine all solutions into one that is still more general than (θ,π). Let $\pi_1 = \pi_{11} \sqcap \pi_{12} \sqcap \pi_{13} \sqcap \pi_{14}$, we have $\pi \leq \pi_1$ based on Lemma 17.

We also need to combine θ_{1i}s. We illustrate the idea by combining θ_{11} and θ_{12} into θ_a. If $\alpha \mapsto V_a \in \theta_{11}$ and $\alpha \notin dom(\theta_{12})$, then we add $\alpha \mapsto V_a$ to θ_a. Dually, if $\alpha \mapsto V_b \in \theta_{12}$ and $\alpha \notin dom(\theta_{11})$, we add $\alpha \mapsto V_b$ to θ_a. If $\alpha \mapsto V_a \in \theta_{11}$ and $\alpha \mapsto V_b \in \theta_{12}$, then we unify V_a and V_b and add the unified result to θ_a. Since both θ_{11} and θ_{12} are more general than θ, θ_a is also more general than θ. Following this idea, let θ_1 be the union of all θ_{11}, θ_{12}, θ_{13}, and θ_{14}, then θ_1 is more general than θ.

Since θ_1 is more general than θ, we have $M_1' \preceq \theta_1(M_1)$. Based on Lemma 13, we have $cod(M_1') \preceq cod(\theta_1(M_1))$, which implies that $M_3' \preceq \theta_1(M_3)$.

Case If: Similar to App.

□

A.4 Proofs of Theorems 13 and 14

Proof of Theorem 13

We start by observing that the auxiliary functions *merge* and *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), $\mathscr{U}$ terminates or makes a call to *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 *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 $\mathscr{U}(M_1 \approx^? M_2)$.

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 $\theta(\alpha) = M$, $\theta(\alpha) \approx_\top \theta(M)$ is clearly satisfied.
- Assume $(\theta, \pi) = \mathscr{U}(d\langle\alpha, \alpha\rangle \approx^? M)$. By the induction hypothesis, $\theta(d\langle\alpha, \alpha\rangle) \approx_\pi \theta(M)$. For any δ such that $\lfloor\theta(d\langle\alpha, \alpha\rangle)\rfloor_\delta \in G$, we have $\lfloor\theta(d\langle\alpha, \alpha\rangle)\rfloor_\delta = \lfloor\theta(\alpha)\rfloor_\delta$. Thus, based on Theorems 1 and 2, Lemmas 9 and 10, and the definition of pattern-constrained relations in Figure 9, we have $\forall\delta.\lfloor\pi\rfloor_\delta = \top \Rightarrow \lfloor\theta(\alpha)\rfloor_\delta = \lfloor\theta(d\langle\alpha, \alpha\rangle)\rfloor_\delta \approx \lfloor\theta(M)\rfloor_\delta$. Now, based on Lemma 11 and pattern-constrained relations, we have $\theta(\alpha) \approx_\pi \theta(M)$, completing the proof for this subcase.

- As $(\theta_1, \pi_1) = \mathscr{U}(\alpha \approx^? \kappa_1 \to \kappa_2)$ and $(\theta_2, \pi_2) = \mathscr{U}(\kappa_1 \to \kappa_2 \approx^? M_1 \to M_2)$, by the induction hypothesis, we have

$$\theta_1(\alpha) \approx_{\pi_1} \theta_1(\kappa_1 \to \kappa_2) \tag{A 3}$$

$$\theta_2(\kappa_1 \to \kappa_2) \approx_{\pi_2} \theta_2(M_1 \to M_2) \tag{A 4}$$

  Moreover, π_1 is $\top$ and θ_1 does not contain mappings for κ_1 and κ_2 as they are fresh. Similarly, θ_2 does not contain a mapping for α since it does not appear in $M_1 \to M_2$. We show that $(\theta_2 \circ \theta_1)(\alpha) \approx_{\pi_2} (\theta_2 \circ \theta_1)(M_1 \to M_2)$ as follows.

$$(\theta_2 \circ \theta_1)(\alpha) = \theta_2(\theta_1(\alpha)) = \theta_2(\kappa_1 \to \kappa_2)$$

$$\begin{aligned} (\theta_2 \circ \theta_1)(M_1 \to M_2) &= \theta_2(\theta_1(M_1 \to M_2)) \\ &= \theta_2(M_1 \to M_2) \\ &\approx_{\pi_2} \theta_2(\kappa_1 \to \kappa_2) && \text{by (4) above} \end{aligned}$$

- The proof is trivial since π is $\bot$.

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

$$\theta_1(M_1) \approx_{\pi_1} \theta_1(M_3) \qquad \theta_2(M_2) \approx_{\pi_2} \theta_2(M_4)$$

Let $\theta' = merge(d, \theta_1, \theta_2)$. We need to show: $\theta'(d\langle M_1, M_2\rangle) \approx_{d\langle \pi_1, \pi_2\rangle} \theta'(d\langle M_3, M_4\rangle)$. By Lemma 11, two types are compatible, if any selection on the two types yields compatible types. Consequently, let's consider selecting $d.1$ on both types in the compatibility relation. We aim to derive the following:

$$\lfloor \theta'(d\langle M_1, M_2\rangle)\rfloor_{d.1} \approx_{\lfloor d\langle \pi_1, \pi_2\rangle\rfloor_{d.1}} \lfloor \theta'(d\langle M_3, M_4\rangle)\rfloor_{d.1}$$

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

$$\lfloor \theta_1(M_1)\rfloor_{d.1} \approx_{\lfloor \pi_1\rfloor_{d.1}} \lfloor \theta_1(M_3)\rfloor_{d.1}$$

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 $\theta'(d\langle M_1, M_2\rangle) \approx_{d\langle \pi_1, \pi_2\rangle} \theta'(d\langle M_3, M_4\rangle)$.

Case (d) and (d^*): Assume we have $(\theta, \pi) = \mathscr{U}(d\langle M_1, M_2\rangle \approx^? d\langle \lfloor M\rfloor_{d.1}, \lfloor M\rfloor_{d.2}\rangle)$. By the induction hypothesis, we have: $\theta(d\langle M_1, M_2\rangle) \approx_\pi \theta(d\langle \lfloor M\rfloor_{d.1}, \lfloor M\rfloor_{d.2}\rangle)$. Our goal is to show:

$$\theta(d\langle M_1, M_2\rangle) \approx_\pi \theta(M)$$

First, for any δ such that $\lfloor \theta(d\langle \lfloor M\rfloor_{d.1}, \lfloor M\rfloor_{d.2}\rangle)\rfloor_\delta \in G$, we have $\lfloor \theta(d\langle \lfloor M\rfloor_{d.1}, \lfloor M\rfloor_{d.2}\rangle)\rfloor_\delta = \lfloor \theta(M)\rfloor_\delta$ 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 $\forall \delta. \lfloor \pi\rfloor_\delta = \top \Rightarrow \lfloor \theta(d\langle M_1, M_2\rangle)\rfloor_\delta \approx \lfloor \theta(d\langle \lfloor M\rfloor_{d.1}, \lfloor M\rfloor_{d.2}\rangle)\rfloor_\delta = \lfloor \theta(M)\rfloor_\delta$. Now, based on Lemma 11 and pattern-constrained relations, we have $\theta(d\langle M_1, M_2\rangle) \approx_\pi \theta(M)$, completing the proof for this case.

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