Solver-based Gradual Type Migration

LUNA PHIPPS-COSTIN, University of Massachusetts Amherst, United States
CAROLYN JANE ANDERSON, Wellesley College, United States
MICHAEL GREENBERG, Pomona College, United States
ARJUN GUHA, Northeastern University, United States

Gradually typed languages allow programmers to mix statically and dynamically typed code, enabling them to incrementally reap the benefits of static typing as they add type annotations to their code. However, this type migration process is typically a manual effort with limited tool support. This paper examines the problem of *automated type migration*: given a dynamic program, infer additional or improved type annotations.

Existing type migration algorithms prioritize different goals, such as maximizing type precision, maintaining compatibility with unmigrated code, and preserving the semantics of the original program. We argue that the type migration problem involves fundamental compromises: optimizing for a single goal often comes at the expense of others. Ideally, a type migration tool would flexibly accommodate a range of user priorities.

We present TypeWhich, a new approach to automated type migration for an extension of the gradually-typed lambda calculus. Unlike prior work, which relies on custom solvers, TypeWhich produces constraints that can be solved by an off-the-shelf MaxSMT solver. This allows us to easily express objectives, such as minimizing the number of necessary syntactic coercions, and constraining the type of the migration to be compatible with unmigrated code.

We present the first comprehensive evaluation of GTLC type migration algorithms, and compare TypeWhich to four other tools from the literature. Our evaluation uses prior benchmarks, and a new set of "challenge problems". Moreover, we design a new evaluation methodology that highlights the subtleties of gradual type migration. In addition, we apply TypeWhich to a suite of benchmarks for Grift, a programming language based on the GTLC. TypeWhich is able to reconstruct all human-written annotations on all but one program.

1 INTRODUCTION

Gradually typed languages allow programmers to freely mix statically and dynamically typed code. This enables users to add static types gradually, providing the benefits of static typing without requiring the entirety of a codebase to be overhauled at once [Siek and Taha 2006; Tobin-Hochstadt and Felleisen 2006]. Over the past decade, gradually typed dialects of several mainstream languages, such as JavaScript, Python, and Ruby, have become established in industry. However, the process of *migrating* an untyped program to use gradual types has largely remained a labor-intensive manual effort. Just as type inference facilitates static typing, type migration tools have the potential to make gradual typing easier to use.

However, automating type migration is a challenging problem. Even if we consider a small language, such as the *gradually typed lambda calculus* (GTLC) [Siek and Taha 2006], and limit ourselves to modifying existing type annotations, a single program may have many possible migrations. Existing approaches either produce a single migration [Flanagan et al. 1996; Henglein and Rehof 1995; Rastogi et al. 2012; Siek and Vachharajani 2008; Wright and Cartwright 1997], or a menu of possible migrations without guidance on which to select [Campora et al. 2018b; Migeed and Palsberg 2020]. How should we choose among the migrations produced by various approaches?

This paper argues that there is a fundamental tension between type migrations that produce precise or "informative" type annotations, and those that preserve the behavior of the original program. In fact, in many GTLC programs, making types more precise can introduce new dynamic errors. Making types more precise can also introduce static and dynamic errors at the (higher-order) boundary between migrated and unmigrated code, a serious concern when migrating a library or a fragment of a larger program. A general-purpose type migration tool would allow programmers to make an informed choice between multiple migrations depending on their context of use.

With these design constraints in mind, we present TypeWhich, a type migration tool that is novel in two key ways. First, unlike prior systems that rely on custom constraint solvers, TypeWhich generates constraints for an off-the-shelf MaxSMT solver [Bjørner et al. 2015]. This makes it easy to add constraints and language features, as we demonstrate by extending the GTLC in several ways and supporting the Grift gradually typed language [Kuhlenschmidt et al. 2019].

Second, using a general-purpose solver allows TypeWhich to support multiple migrations with different properties: the user can prioritize migrations with the most informative types, or migrations that maximize compatibility with unmigrated code, or something in between. We accomplish this by using the MaxSMT solver in a two-stage process. We first formulate a MaxSMT problem with an objective function that synthesizes precise types. The reconstructed type of the program may not be compatible with all contexts, but it reveals the (potentially higher-order) interface of the program. We then formulate new constraints on the type of the program to enforce compatibility, and use the MaxSMT solver a second time to produce a new solution.

This paper also presents the first comprehensive evaluation of different type migration approaches. We compare TypeWhich to four other type migration approaches using a two-part evaluation suite: a set of existing benchmarks by Migeed and Palsberg [2020], and a new set of "challenge problems" that we devise. We also design an evaluation methodology that reflects the subtleties of type migration. Although different approaches to type migration prioritize different goals, TypeWhich performs well in our evaluation. An advantage of TypeWhich over most existing work is that it does not reject any untyped programs. Finally, we apply TypeWhich to a suite of Grift programs from Kuhlenschmidt et al. [2019], and find that it reproduces all hand-written type annotations except in one case.

Contributions. Our key contributions are as follows:

(1) We characterize the many goals of type migration, illustrate their inherent competition, and argue that type migration tools should allow users to make informed decisions about their own priorities (§2 and §3).
(2) We present the TypeWhich approach to type migration, which formulates constraints for an off-the-shelf MaxSMT solver (§4). TypeWhich supports the GTLC and additional language features required to support the Grift gradually typed language (§5).
(3) We present a new set of type migration "challenge problems" that illustrate the strengths and weaknesses of different approaches to type migration (§6).
(4) We present a comprehensive comparison of five approaches to type migration (including ours), using a new evaluation methodology. For this comparison, we implement a unified framework for running, evaluating, and validating type migration algorithms.
(5) Finally, we contribute re-implementations of the type migration algorithms from Campora et al. [2018b] and Rastogi et al. [2012]. Ours is the first publicly available implementation of Rastogi et al. [2012].

2 WHAT MATTERS FOR TYPE MIGRATION?

When designing a type migration tool, we must consider several important questions:

(1) A key goal of type migration is to improve the precision of type annotations. However, there are often multiple ways to improve type precision [Migeed and Palsberg 2020] that induce different run-time checks. For any given type migration system, we must therefore ask the question, *Can a user choose between several alternative migrations?*
(2) When the migrated code is only a fragment of a larger codebase, increasing type precision can introduce type errors at the boundaries between migrated and unmigrated code [Rastogi et al.

2012]. Thus we must ask, *Does the migrated code remain compatible with other, unmigrated code?*

(3) A type migration tool may also uncover potential run-time errors. However, these errors may be unreachable, or only occur in certain configurations or on certain platforms. Thus we must ask, *Should a migration turn (potential) run-time errors into static type errors?*

(4) Finally, safe gradually typed languages introduce checks that enforce type safety at run-time. Making a type more precise can alter these checks, affecting run-time behavior. Thus we must ask, *Does the migrated program preserve the behavior of the original program?*

This section explores these questions with examples from the gradually-typed lambda calculus (GTLC) with some modest extensions. We write programs in an OCaml-like syntax with explicit type annotations. The type ★ is the *unknown type* (also known as the dynamic type or the any type), which is compatible with all types. Under the hood, converting to and from the ★ type introduces coercions [Henglein 1994]; these coercions can fail at run-time with a dynamic type error.

Type migration can introduce new static errors. Figure 1 shows a function that uses its ★-typed argument first as a number and then as a function. Since ★ is compatible with all types, the function is well-typed, but guaranteed to produce a dynamic type error when applied. In this case, it seems harmless for a type migration tool to turn this dynamic type error into a static type error.

```
let A (x : ★) =
  let _ = x + 10 in
  x ()
```

Fig. 1. Reachable error.

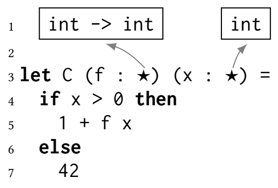
However, it is also possible for the crashing expression to be unreachable. Figure 2 wraps the same dynamic error in the unused branch of a conditional. In this case, improving the type annotation would lead to a spurious error: the migrated program would fail even though the original ran without error. Although this example is contrived, programs in untyped languages often have code whose reachability is environment-dependent (e.g., JavaScript web programs that support multiple browsers, Python programs that can be run in Python 2 and 3). The flexibility of gradual typing is particularly valuable in these cases, but reasoning about safety and precision in tandem is subtle.

```
let B (x : ★) =
  if false then
    let _ = x + 10 in
    x ()
  else
    x ()
```

Fig. 2. Unreachable error.

Type migration can restrict the context of a program. There are many cases where it is impractical to migrate an entire program at once. For example, the programmer may not be able to modify the source code of a library; they may be migrating a library that is used by others; or it may just be unacceptable to change every file in a large software project. In these situations, the type migration question is even trickier.

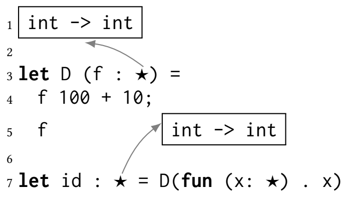
Figure 3 shows a higher-order function C that calculates $1 + f(x)$ when x is greater than zero. We could migrate C to require f to be an integer function, which precisely captures how C uses f. However, this migration makes some calls to C ill-typed. For example, `C 0 0` evaluates to `42` before migration, but is ill-typed after migration.



```
 int -> int      int

let C (f : ★) (x : ★) =
  if x > 0 then
    1 + f x
  else
    42
```

Fig. 3. Context restriction.

Figure 4 illustrates another subtle interaction between type-migrated code and its context. The function D receives f and expects it to be a function over numbers. Unlike the previous example, D always calls f, so it may appear safe to annotate f with the type `int -> int`. However, D also returns f back to its caller, so this migration changes the return type of D from ★ to `int -> int`,. For example, when f is the identity function, $D(f)$



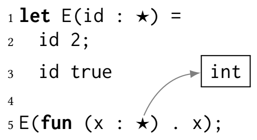
```
int -> int

let D (f : ★) =
  f 100 + 10;
  f                int -> int

let id : ★ = D(fun (x: ★) . x)
```

Fig. 4. Context restriction.

returns the identity function before migration, but after migration $D(f)$ is restricted to only work on `int`s.

To summarize, there is a fundamental trade-off between making types precise in migrated code, and maintaining compatibility with unmigrated code.

Type migration can introduce new dynamic errors. So far, we have looked at migrations that introduce static type errors. However, there is a more insidious problem that can occur: a migration can introduce new dynamic type errors. Figure 5 shows a program that runs without error: `E` receives the identity function and applies it to two different types. However, since `E`'s argument has type ★, which is compatible with all types, the program is well-typed even if we migrate the identity function to require an integer argument. Gradual typing will wrap the function to dynamically check that it only receives integers. So the program runs without error before migration, but produces a dynamic type error after migration. Strictly speaking, although this migration introduces a new dynamic error, its static types are more precise. When evaluating migrations, it is not enough to consider just the types or interfaces: it is important to understand which run-time checks will be inserted.



```
let E(id : ★) =
  id 2;
  id true

E(fun (x : ★) . x);
```

Fig. 5. Dynamic type error.

In summary, there are several competing concerns that we must consider when choosing an approach to type migration. TypeWhich prioritizes preserving the behavior of the original program: it produces types that do not introduce new static or dynamic errors in the migrated code. However, this objective leaves the question of context unanswered. Should TypeWhich produce the most precise type it can? This may make the migrated code incompatible with unmigrated code. So, should TypeWhich instead produce a type that is compatible with all untyped code? This would mean discarding a lot of useful information, e.g., the types of function arguments. Or, should TypeWhich strike a compromise between precision and compatibility? We think the right answer depends on the context in which the type migration tool is being used. Instead of making an arbitrary decision, TypeWhich allows the programmer to choose between several migrations that prioritize different properties.

3 FORMALIZING THE TYPE MIGRATION PROBLEM

We now formally define the type migration problem. We first briefly review the *gradually typed lambda calculus* (GTLC) [Siek et al. 2015b], which is a core calculus for mixing typed and untyped code. We then present several definitions of type migration for the GTLC.

3.1 The Gradually Typed Lambda Calculus

The Gradually Typed Lambda Calculus (GTLC) extends the typed lambda calculus with base types (integers and booleans) and the *unknown type* ★. Figure 6 shows its syntax and typing rules.

Type checking relies on the *type consistency* relation, $S \sim T$. Type consistency determines whether an S-typed expression may appear in a T-typed context. Two types are consistent if they are structurally equal up to any unknown (★) types within them; the ★-type is consistent with all types and any expression may appear in a ★-typed context. The type consistency relation is reflexive and symmetric, but not transitive: int and bool are both consistent with ★ but not with each other.

The typing rules for identifiers, literals, and functions are straightforward, but there are two function application rules: (1) If the expression in function position has type ★, then the argument may have any type, and the result of the application has type ★. (2) When the type of the function

Base types

$B := \mathsf{int} \mid \mathsf{bool}$

Types and contexts

$S, T := B$	Base type
$\quad \mid S \to T$	Function type
$\quad \mid \star$	Unknown type
$\Gamma := \cdot \mid \Gamma, x : T$	

Constants

$b := \mathsf{true} \mid \mathsf{false}$	Boolean literal
$n := \cdots$	Integer literal
$c := b \mid n$	

Expressions

$e := x$	Identifier
$\quad \mid c$	Literal
$\quad \mid \mathsf{fun}(x : T).e$	Function
$\quad \mid e_1\ e_2$	Application
$\quad \mid e_1 \times e_2$	Multiplication

Type Consistency $\boxed{T \sim T}$

$$\frac{}{\star \sim T} \qquad \frac{}{T \sim \star} \qquad \frac{}{T \sim T} \qquad \frac{}{B \sim B} \qquad \frac{S_1 \sim S_2 \qquad T_1 \sim T_2}{S_1 \to T_1 \sim S_2 \to T_2}$$

Typing Literals $\boxed{ty : c \to B}$

$ty(n) = \mathsf{int} \quad ty(b) = \mathsf{bool}$

Typing $\boxed{\Gamma \vdash e : T}$

$$\frac{\Gamma(x) = T}{\Gamma \vdash x : T} \qquad \frac{}{\Gamma \vdash c : ty(c)} \qquad \frac{\Gamma, x : S \vdash e : T}{\Gamma \vdash \mathsf{fun}(x : S).e : S \to T}$$

$$\frac{\Gamma \vdash e_1 : \star \qquad \Gamma \vdash e_2 : T}{\Gamma \vdash e_1\ e_2 : \star} \qquad \frac{\Gamma \vdash e_1 : S \to T \qquad \Gamma \vdash e_2 : S' \qquad S \sim S'}{\Gamma \vdash e_1\ e_2 : T}$$

$$\frac{\Gamma \vdash e_1 : S \qquad \Gamma \vdash e_2 : T \qquad S \sim \mathsf{int} \qquad T \sim \mathsf{int}}{\Gamma \vdash e_1 \times e_2 : \mathsf{int}}$$

Fig. 6. The Gradually Typed Lambda Calculus (GTLC): surface syntax and typing.

expression is an arrow type ($S \to T$), the result has type T. The type of the argument must be consistent with—but not necessarily equal to—the type of argument the function expects ($S' \sim S$).

We add a built-in multiplication operator that requires the types of its operands to be consistent with int (i.e., an operand may have type $\star$). We choose multiplication because the "+" operator is overloaded in many untyped languages: we add addition in §5, where we discuss overloading.

3.2 Ground Types and Coercion-based Semantics

Programs in the GTLC are not run directly, but are first compiled to an intermediate representation where static type consistency checks are turned into dynamic checks if necessary. There are two well-known mechanisms for describing these dynamic checks: casts and coercions. We use coercions, following Henglein [1994], as they most closely match the type-tagging and tag-checking operations used at run-time in dynamic languages.[1]

The *ground types* (G in Figure 7) are the types that are dynamically observable, and include all base types and a ground type fun for all functions. The two basic coercions (k) tag a value with a ground type ($G!$) and untag a value after checking that it has a particular ground type ($G?$). Both of these operations can fail: an already-tagged value cannot be re-tagged, and untagging succeeds only if the value has the expected ground type. There are three additional coercions: identity coercions, which exist only to simplify certain definitions; a sequencing coercion ($k_1; k_2$); and a *function proxy* wrap that lifts coercions to functions.

To see how the coercion system works, consider a case where we have a $\star$-typed value f that we want to treat as a function of type $\mathsf{int} \to \mathsf{int}$. To do so, we apply f to a coercion as follows:

$$[\mathsf{fun?}; \mathsf{wrap}(\mathsf{int!}, \mathsf{int?})]f$$

This coercion first checks that that f is a function (fun?), and then wraps f in a function proxy that will tag its int argument (since f expects a $\star$ value) and will untag its result (since f returns a $\star$, but we expect an int).

The values of the language (v) include constants, functions, and values tagged with a ground type. We define tagged values ($\mathsf{box}(G, u)$) so that a tag can only be placed on an untagged value (u).

[1]The two approaches are inter-translatable [Greenberg 2013; Herman et al. 2011] with full abstraction [Siek et al. 2015a].

Ground types

$G ::= B \mid \mathsf{fun}$

Coercions

$k ::= G?$	Untag
$\mid G!$	Tag
$\mid \mathsf{wrap}(k_1, k_2)$	Wrap function
$\mid k_1; k_2$	Sequence
$\mid \mathsf{id}_T$	Identity

Expressions

$e ::= \cdots \mid [k]\ e$	Apply coercion

Untagged values

$u ::= c \mid \mathsf{fun}(x : T).e$

Values

$v ::= u \mid \mathsf{box}(G, u)$

Evaluation Contexts

$E ::= [] \mid E\ e \mid v\ E \mid [k]\ E$

Active Expressions

$ae ::= (\mathsf{fun}(x : T).e)\ v \mid [k]\ v$

Evaluation $\boxed{\vdash e \hookrightarrow e}$

$$
\begin{array}{l}
(\mathsf{fun}(x : T).e)\ v \hookrightarrow e[x/v] \\
[\mathsf{id}]\ v \hookrightarrow v \\
[G!]\ (u) \hookrightarrow \mathsf{box}(G, u) \\
[G?]\ (\mathsf{box}(G, u)) \hookrightarrow u \\
[\mathsf{wrap}(k_1, k_2)]\ v \\
\hookrightarrow \mathsf{fun}(x : \star).[k_2]\ (v\ ([k_1]\ x)) \\
[k_1; k_2]\ v \hookrightarrow [k_2]\ ([k_1]\ v)
\end{array}
$$

$$\frac{ae \hookrightarrow e'}{E[ae] \hookrightarrow E[e']}$$

$$
\begin{array}{l}
\mathsf{coerce}(T, T) = \mathsf{id}_T \\
\mathsf{coerce}(\star, b) = b? \\
\mathsf{coerce}(b, \star) = b! \\
\mathsf{coerce}(\star, \star \to \star) = \mathsf{fun}? \\
\mathsf{coerce}(\star \to \star, \star) = \mathsf{fun}! \\
\mathsf{coerce}(S_1 \to S_2, T_1 \to T_2) = \mathsf{wrap}(\mathsf{coerce}(T_1, S_1), \mathsf{coerce}(S_2, T_2)) \\
\mathsf{coerce}(\star, T_1 \to T_2) = \mathsf{fun}?; \mathsf{wrap}(\mathsf{coerce}(\star, T_1, \star), \mathsf{coerce}(\star, T_2)) \\
\mathsf{coerce}(T_1 \to T_2, \star) = \mathsf{wrap}(\mathsf{coerce}(\star, T_1), \mathsf{coerce}(T_2, \star)); \mathsf{fun}! \\
\mathsf{coerce}(S, T) = \mathsf{coerce}(S, \star); \mathsf{coerce}(\star, T)
\end{array}
$$

Coercion Insertion $\boxed{\Gamma \vdash e \Rightarrow e, T}$

$$\frac{\Gamma(x) = T}{\Gamma \vdash x \Rightarrow x, T} \qquad \frac{}{\Gamma \vdash c \Rightarrow c, ty(c)}$$

$$\frac{\Gamma, x : S \vdash e \Rightarrow e', T}{\Gamma \vdash \mathsf{fun}(x : S).e \Rightarrow \mathsf{fun}(x : S).e', S \to T}$$

$$\frac{\Gamma \vdash e_1 \Rightarrow e_1', S \to T \qquad \Gamma \vdash e_2 \Rightarrow e_2', S'}{\Gamma \vdash e_1\ e_2 \Rightarrow e_1'\ ([\mathsf{coerce}(S', S)]\ e_2'), T}$$

$$\frac{\Gamma \vdash e_1 \Rightarrow e_1', T \qquad T \neq T_1 \to T_2 \qquad \Gamma \vdash e_2 \Rightarrow e_2', S}{\Gamma \vdash e_1\ e_2 \Rightarrow ([\mathsf{coerce}(T, \star \to \star)]\ e_1')\ ([\mathsf{coerce}(S, \star)]\ e_2'), \star}$$

Fig. 7. Coercion insertion and evaluation for the GTLC.

Type Precision $\boxed{T \sqsubseteq T}$

$$\frac{}{\star \sqsubseteq T} \qquad \frac{}{T \sqsubseteq T} \qquad \frac{S_1 \sqsubseteq S_2 \qquad T_1 \sqsubseteq T_2}{S_1 \to T_1 \sqsubseteq S_2 \to T_2}$$

Expression Precision $\boxed{e \sqsubseteq e}$

$$\frac{}{x \sqsubseteq x} \qquad \frac{}{c \sqsubseteq c} \qquad \frac{e_1 \sqsubseteq e_1' \qquad e_2 \sqsubseteq e_2'}{e_1\ e_2 \sqsubseteq e_1'\ e_2'} \qquad \frac{T \sqsubseteq T' \qquad e \sqsubseteq e'}{\mathsf{fun}(x : T).e \sqsubseteq \mathsf{fun}(x : T').e'}$$

Fig. 8. Type and expression precision.

The coercion insertion rules are analogous to typing, but produce both a type and an equivalent expression with explicit coercions. They rely on the *coerce* metafunction that translates a static consistency check $S \sim T$ into a corresponding coercion that is dynamically checkable. When two types are identical, *coerce* produces the identity coercion, which can be safely removed. The final case of *coerce* addresses inconsistencies ($S \nsim T$). Instead of rejecting programs with inconsistent checks, we produce a coercion that is *doomed to fail*. Gradual typing systems sometimes reject programs that demand casts between incompatible types. However, doing so violates the desired property that migrations should preserve the behavior of the original program when possible. If we rejected these programs, a user would need to excise all incompatibilities, whether or not they are in live code branches, at the onset of migration.

3.3 Type Migration

All formulations of the type migration problem rely on defining *type precision*, where $\star$ is the least precise type. The type precision relation (Figure 8), written $S \sqsubseteq T$, is a partial order that holds when S is less precise than T (or S and T are identical). We use type precision to define expression precision in the obvious way: an expression is more precise than its structural equivalent if its type annotations are more precise according to the type precision relation.

Migeed and Palsberg [2020] define a type migration as an expression that has more precise type annotations, and use this definition to study the decidability and computational complexity of several problems, such as finding migrations that cannot be made more precise.

Definition 3.1 (Type Migration). Given $\vdash e : T$ and $\vdash e' : T'$, e' is a *type migration* of e if $e \sqsubseteq e'$ and $T \sqsubseteq T'$.

However, as we argued in §2, improving type precision is one of several competing goals for type migration. Another important goal is to preserve the behavior of the original program. To reason about this, we must reformulate the definition of a type migration to relate the values produced by the original expression and its migration. We propose the following definition of value-restricted type migration:

Definition 3.2 (Value-restricted Type Migration). Given $\vdash e : T$ and $\vdash e' : T'$, e' is a *restricted type migration* of e if:

(1) $e \sqsubseteq e'$;
(2) $T \sqsubseteq T'$; and
(3) $e \hookrightarrow^* v$ if and only if $e' \hookrightarrow^* v'$ with $v \sqsubseteq v'$.

This definition of type migration relates the values of the two expressions. However, this definition is too weak. For one thing, it does not say anything about programs that produce errors or do not terminate. But there is a more serious problem: it is too permissive for function types. For example, given the identity function with type $\star \rightarrow \star$, this definition allows a type migration that changes its type to $\mathsf{int} \rightarrow \mathsf{int}$, which will produce a dynamic type error if the function is applied to non-integers.

To address this issue, the definition of type migration must take into account the contexts in which the migrated expression is used. We define a well-typed program context C as a context with a hole that can be filled with a well-typed open expression to get a well-typed closed expression.

Definition 3.3 (Well-Typed Program Context). A program context C is well typed, written $C : (\Gamma \vdash S) \Rightarrow T$ if for all expressions e where $\Gamma \vdash e : S$ we have $\vdash C[e] : T$.

We now define a *context-restricted type migration* as a more precisely-typed expression that is equivalent to the original expression in all contexts that can be filled with an expression of a given type S. Note that the type expected by the context (S) must be consistent (but not identical) with the types of both the original and the migrated expression.

Definition 3.4 (Context-restricted Type Migration). Given $\vdash e : T$, $\vdash e' : T'$, and a type S where $S \sim T$ and $S \sim T'$, e' is a *context-restricted type migration* of e at type S if:

(1) $e \sqsubseteq e'$;
(2) $T \sqsubseteq T'$; and
(3) For all $C : (\cdot \vdash S) \Rightarrow U$, either a) $C[e] \hookrightarrow^* v$ and $C[e'] \hookrightarrow^* v'$ with $v \sqsubseteq v'$; b) both $C[e]$ and $C[e']$ get stuck at a failed coercion; or c) both $C[e]$ and $C[e']$ do not terminate.

At the limit, the context's expected type S could be $\star$, in which case the definition is essentially equivalent to that of Rastogi et al. [2012, Theorem 3.22]. However, this is a very strong requirement that rules out many informative migrations (§2). If the programmer is comfortable making assumptions about how the rest of the program will interact with the migrated expression, they may choose a more precise S, and allow a wider range of valid type migrations.

Types

$T ::= \cdots$
$\quad\mid \alpha, \beta, \gamma, \delta$ — Type metavariables

Coercions

$k ::= \cdots$
$\quad\mid \underline{coerce}(T_1, T_2)$ — Coercion from T_1 to T_2

Type Representation

```
(declare-datatypes ()
  ((Typ (star) (int) (bool)
        (arr (in Typ) (out Typ)))))
```

Constraints

$\phi ::= T_1 = T_2$ — Type equality
$\quad\mid w$ — Boolean variable (weight)
$\quad\mid \phi_1 \wedge \phi_2$ — Conjunction
$\quad\mid \phi_1 \vee \phi_2$ — Disjunction
$\quad\mid \neg\phi$ — Negation

Constraint Metafunctions

$ground \in T \to \phi$
$ground(T) = T \in B \vee T = \star \to \star$

Fig. 9. The type constraint language, and language extensions for constraint generation.

4 THE TYPEWHICH APPROACH TO TYPE MIGRATION

We now present TypeWhich, an approach to type migration that differs in two ways from previous work. (1) Instead of relying on a custom constraint solver, TypeWhich produces constraints and an objective function for the Z3 MaxSMT solver [Bjørner et al. 2015]. (2) Instead of producing a single migration, or several migrations without guidance on which to choose, TypeWhich allows the user to choose between migrations that prioritize type precision, compatibility with untyped code, or other properties. Moreover, the TypeWhich migration algorithm handles these different scenarios in a uniform, type-directed way. This section presents TypeWhich's type migration algorithm for the core GTLC. §5 extends TypeWhich with additional language features, including some that have not been precisely described in prior work.

4.1 The Language of Type Constraints

For the purpose of constraint generation, we make two additions to the GTLC (Figure 9):

(1) We extend types with type metavariables (α).
(2) We introduce a new coercion, $\underline{coerce}(S, T)$, which represents a suspended call to the *coerce* metafunction (Figure 7). The type arguments to $\underline{coerce}$ may include type metavariables. After constraint solving, we substitute any type metavariables with concrete types and use the *coerce* metafunction to get a primitive coercion (k).

Both of these are auxiliary and do not appear in the final program.

The constraints (ϕ) that we generate are boolean-sorted formulas for a MaxSMT solver that supports the theory of algebraic datatypes [Barrett et al. 2007]. In addition to the usual propositional connectives, our constraints involve equalities between types ($T_1 = T_2$), predicates over types (e.g., to check if a type is an arrow type), and auxiliary boolean variables (w). We use these boolean variables to define soft constraints that guide the solver towards solutions with fewer non-trivial coercions.

Using Z3's algebraic datatypes, we define a new sort (`Typ`) that encodes all types (T) except type metavariables. Constraint generation defines a `Typ`-sorted constant for every metavariable that occurs in a type. For example, we can solve the type constraint $\alpha \to \text{int} = \beta$ with the following commands to the solver:

```
(declare-const alpha Typ)
(declare-const beta Typ)
(assert (= (arr alpha int) beta))
```

This example is satisfiable, and the model assigns `alpha` and `beta` to metavariable-free types (represented as `Typ`). If σ is such a model, we write Subst(σ, β) to mean the metavariable-free type

$$\boxed{\Gamma \vdash e \Rightarrow e, T, \phi}$$

$$\textsc{Id}\ \frac{\phi = (\alpha = \Gamma(x) \wedge w) \vee (\alpha = \star \wedge \neg w) \quad \alpha, w \text{ is fresh}}{\Gamma \vdash x \Rightarrow [\underline{coerce}(\Gamma(x), \alpha)]x, \alpha, \phi} \qquad \textsc{Const}\ \frac{\phi = (\alpha = ty(c) \wedge w) \vee (\alpha = \star \wedge \neg w) \quad \alpha, w \text{ is fresh}}{\Gamma \vdash c \Rightarrow [\underline{coerce}(ty(c), \alpha)]c, \alpha, \phi}$$

$$\textsc{Fun}\ \frac{\begin{array}{c}\Gamma, x : \alpha \vdash e \Rightarrow e', T, \phi_1 \quad \beta, w \text{ fresh} \\ \phi_2 = (\beta = \alpha \to T \wedge w) \vee (\beta = \star \wedge ground(\alpha \to T) \wedge \neg w)\end{array}}{\Gamma \vdash \mathsf{fun}(x : \alpha).e \Rightarrow [\underline{coerce}(\alpha \to T, \beta)]\mathsf{fun}(x : \alpha).e', \beta, \phi_1 \wedge \phi_2}$$

$$\textsc{App}\ \frac{\begin{array}{c}\Gamma \vdash e_1 \Rightarrow e_1', T_1, \phi_1 \qquad \Gamma \vdash e_2 \Rightarrow e_2', T_2, \phi_2 \qquad \alpha, \beta, \gamma, w_1, \text{and } w_2 \text{ are fresh} \\ \phi_3 = (T_1 = \alpha \to \beta \wedge w_1) \vee (T_1 = \alpha = \beta = \star \wedge \neg w_1) \qquad \phi_4 = (T_2 = \alpha) \qquad \phi_5 = (\beta = \gamma \wedge w_2) \vee (\gamma = \star \wedge \neg w_2)\end{array}}{\Gamma \vdash e_1\ e_2 \Rightarrow [\underline{coerce}(\beta, \gamma)](([\underline{coerce}(T_1, \alpha \to \beta)]e_1')\ e_2'), \gamma, \phi_1 \wedge \phi_2 \wedge \phi_3 \wedge \phi_4 \wedge \phi_5}$$

$$\textsc{Mul}\ \frac{\begin{array}{c}\Gamma \vdash e_1 \Rightarrow e_1', T_1, \phi_1 \qquad \Gamma \vdash e_2 \Rightarrow e_2', T_2, \phi_2 \qquad w_1, w_2, \text{and } w_3 \text{ are fresh} \\ \phi_3 = (T_1 = \mathsf{int} \wedge w_1) \vee (T_1 = \star \wedge \neg w_1) \quad \phi_4 = (T_2 = \mathsf{int} \wedge w_2) \vee (T_2 = \star \wedge \neg w_2) \\ \phi_5 = (\alpha = \mathsf{int} \wedge w_3) \vee (\alpha = \star \wedge \neg w_3)\end{array}}{\Gamma \vdash e_1 \times e_2 \Rightarrow [\underline{coerce}(\mathsf{int}, \alpha)]([\underline{coerce}(T_1, \mathsf{int})]e_1' \times [\underline{coerce}(T_2, \mathsf{int})]e_2'), \alpha, \phi_1 \wedge \phi_2 \wedge \phi_3 \wedge \phi_4 \wedge \phi_5}$$

Fig. 10. Constraint generation for GTLC

assigned to β, i.e., the closure of substituting with the model σ. In this example, α is unconstrained, so there are several possible models: $\sigma = \{\alpha \mapsto \mathsf{int}, \beta \mapsto \alpha \to \mathsf{int}\}$ is one, as is $\sigma' = \{\alpha \mapsto \star, \dots\}$. We have $\textsc{Subst}(\sigma, \beta) = \mathsf{int} \to \mathsf{int}$, while $\textsc{Subst}(\sigma', \beta) = \star \to \mathsf{int}$.

Finally, for succinctness, we define $ground(T)$, which produces a constraint that is satisfiable when T is a ground type. At the moment, the only ground types are base types and dynamic function types ($\star \to \star$). §5 extends the language with additional types and augments the definition of *ground*.

4.2 Generating Type Constraints

We now present constraint generation for the GTLC. To simplify the presentation, we assume that all bound variables have type $\star$. Constraint generation is a two-step process:

(1) We replace every $\star$ annotation in the input program with a fresh metavariable. The solution to the constraints maps these metavariables to types, which may be more precise than $\star$.
(2) We generate constraints by applying deterministic, syntax-directed inference rules.

Since the first step is straightforward, we focus on constraint generation. The constraint generation rules are of the form $\Gamma \vdash e \Rightarrow e', T, \phi$: the inputs are the type environment (Γ) and the expression (e), and the outputs are as follows:

(1) An output expression (e') that is equivalent to the input expression, but with explicit coercions.
(2) A type (T), which is the type of the expression, and may include metavariables.
(3) A constraint (ϕ) with type-sorted and boolean-sorted free variables.

When formulating constraint generation, there are several requirements to keep in mind. First, the constraint ϕ may be satisfiable in several ways. We will eventually use soft constraints to choose among solutions, but we design the constraint generation process so that *all models of ϕ correspond to valid migrations*. Second, as argued in §2, we do not want to reject any programs. We therefore set up constraint generation so that we *do not introduce new static errors*. Our final goal is to *favor informative types*. We do this via soft constraints that penalize the number of non-trivial, syntactic coercions. Note that this is not the same as minimizing the number of coercions performed during evaluation, which is a harder problem that we leave for future work (but see Campora et al. [2018a]).

Constraint Generation Rules. Constraint generation is syntax directed (Figure 10). As a general principle, we allow all expressions to be coerced to $\star$: this enables us to migrate all programs, even though it may generate coercions that are doomed to fail if they are ever run. This property is critical to ensure that models exist for all programs (Theorem 4.2).[2]

Following this principle, the rule for identifiers (Id) introduces a coercion that is either the identity coercion (when α is T, the type of the identifier in the environment), or a coercion to $\star$ (when α is $\star$). At a later step (§4.3), we produce a soft constraint favoring w over $\neg w$, which guides the solver towards solutions that avoid the non-trivial coercions when possible.

Similarly, the rule for constants (Const) generates two new variables: α and a fresh weight variable w. The rule constrains the type α to either be the type of the constant, or the $\star$ type (i.e, to avoid rejecting true $\times$ 1). In the former case, we constrain w to be true, and in the latter, to false.

The rule for functions (Fun) assumes that the argument is annotated with a unique metavariable (α) and recurs into the function body, which produces some type T. The rule gives the function the type β (a fresh metavariable), and constrains it to be the type of the function ($\alpha \rightarrow T$) or the $\star$ type. In the latter case, we also constrain the type of the function to be the ground type ($\star \rightarrow \star$). We use a weight w to prefer the former case without rejecting expressions like $1 \times (\mathsf{fun}(x : \star).x)$.

The rule for function applications (App) produces a constraint that is a conjunction of five clauses: ϕ_1 and ϕ_2 are the constraints that arise when recurring into the two sub-expressions of the application; ϕ_3 constraints the type of the function; ϕ_4 constrains the type of the argument; and ϕ_5 constrains the type of the result. Together, ϕ_3 and ϕ_4 capture the two ways in which applications can be typed in the GTLC: the function may be of type $\star$, in which case it is coerced to the function ground type, $\star \rightarrow \star$ and w is false, or the function already has a function type, and w_1 is true. In either case, the argument type is constrained to be the function input type α. The final constraint allows the result type, β, to be coerced to $\star$; w_2 is true only if this is a non-trivial coercion.

The rule for multiplication (Mul) produces a five-part conjunction: ϕ_1 and ϕ_2 are the constraints produced by its operands; ϕ_3 and ϕ_4 constrain each operand to either be int or $\star$ and use weights to prefer the former; and ϕ_5 constrains the type of the result to either be int or $\star$, with a weight that prefers for the former; again, this is necessary to avoid rejecting programs.

Example 1: Types for the Identity Function. Consider the following program, which applies the identity function to 42 and true, and has the least precise type annotations:[3]

$$(\mathsf{fun}(id : \star).(\mathsf{fun}(n : \star).id\ \mathsf{true})(id\ 42))\ (\mathsf{fun}(x : \star).x)$$

First, consider how we might manually migrate the program. One approach is to change the type of x to int (underlined below), and leave the other annotations unchanged:

$$(\mathsf{fun}(id : \star).(\mathsf{fun}(n : \star).id\ \mathsf{true})\ (id\ 42))\ (\mathsf{fun}(x : \underline{\mathsf{int}}).x)$$

It is important to note that this program is well-typed and has a more precise type than the original. However, it produces a run-time type error on *id true*, whereas the original program does not. Fortunately, constraint generation rules out this migration: the outermost application coerces the argument type to $\star$. However, the argument type ($\mathsf{int} \rightarrow \mathsf{int}$) is not a ground type, which App also requires. Thus our constraint generation algorithm rules out this migration.

[2] We have also implemented a version of TypeWhich that uses an alternative constraint generation rule for identifiers that enforces rigid types together with a modified version of the function application rule that can coerce the function argument. This leads to a loss of type precision, but produces type annotations that are more robust to code-refactoring. Both approaches are sound and safe at the generated types (§4.3).

[3] This is a variation of the example in Figure 5.

```
 1: ▷ The only annotations in $e$ are $\star$
 2: function PreciseMigrate(e)
 3:     $e_1 \leftarrow$ IntroduceMetavars($e$)            ▷ Replace every $\star$ with a fresh $\alpha$s
 4:     $\cdot \vdash e \Rightarrow e', T_1, \phi$          ▷ Generate constraints and objectives
 5:     for $\alpha \in \phi$ do                            ▷ The set of type metavariables in $\phi$
 6:         (declare-const $\alpha$ Typ)
 7:     for $w \in \phi$ do                                 ▷ The set of weight variables in $\phi$
 8:         (declare-const $w$ Bool)
 9:         (assert-soft $w$ 1)
10:     (check-sat $\phi$)
11:     $\sigma \leftarrow$ (get-model)                     ▷ Model mapping type metavariables to types
12:     return Subst($\sigma, e'$)                          ▷ Migrated program without explicit coercions
```

Fig. 11. Precise Type Migration.

The following type migration, also constructed manually, is the most precise migration that does not introduce a run-time error (changes to the original program are underlined):

$$(\mathsf{fun}(id : \underline{\star \to \star}).(\mathsf{fun}(n : \underline{\mathsf{int}}).id\ \mathsf{true})\ (id\ 42))\ (\mathsf{fun}(x : \star).x)$$

However, concluding that n has type int requires reasoning about the flow of values through the identity function. Our constraint generation rules can't find this solution. Instead, the most precise type allowed by our constraints gives id the type $\star \to \star$ and leaves n and x at type $\star$:

$$(\mathsf{fun}(id : \underline{\star \to \star}).(\mathsf{fun}(n : \star).id\ \mathsf{true})\ (id\ 42))\ (\mathsf{fun}(x : \star).x)$$

This example illustrates an important principle that we follow in constraint generation: if we generate a new coercion around an expression e to type $\star$, then we must also *constrain* the type of e to be a ground type. As we grow the language with more types, the set of ground types will grow. When this happens, we update the definition of the *ground* predicate, but the rest of constraint generation remains unchanged.

The following theorem establishes that all models that satisfy our constraint generation rules produce well-typed expressions.

Theorem 4.1 (Type Migration Soundness). *If $\Gamma \vdash e \Rightarrow e', T, \phi$ and σ is a model for ϕ, then $\textsc{Subst}(\sigma, \Gamma) \vdash \textsc{Subst}(\sigma, e') : \textsc{Subst}(\sigma, T)$.*

Proof. By induction on the coercion insertion judgment (see Theorem C.2 for more details). □

4.3 Solving Constraints for Precise Type Migration

Our formulation of constraint generation produces a constraint (ϕ) that may have multiple models, all of which encode valid type migrations of varying precision. Our goal in this section is to find as precise a migration as possible. To do this, we rely on the MaxSMT solver's ability to define *soft constraints*. The solver prefers solutions that obey these constraints, but can violate them when necessary to produce a model.

Our constraint generation rules adhere to the following recipe: every rule that introduces a coercion also introduces a fresh boolean variable (w) that is true when the coercion is trivial ($coerce(T, T)$) and false otherwise. The Fun rule introduces one boolean variable, while the App rule introduces two, since it may introduce two non-trivial coercions.

We use the algorithm sketched in Figure 11. For each boolean variable, we produce a soft constraint asserting that w should hold (the corresponding coercion should be trivial if possible). Given these soft constraints, we check that the formula ϕ is satisfiable and get a model (σ) that assigns type metavariables to types. We then substitute metavariables with concrete types accordingly.

Example 2: A migration that is too precise. Consider the following program as an input to our algorithm:

$$F_1 \triangleq \mathsf{fun}(f : \star).\mathsf{fun}(g : \star).(f\ 1) \times (g\ f)$$

The algorithm produces the following migration, which has the most precise types possible:

$$F_2 \triangleq \mathsf{fun}(f : \mathsf{int} \rightarrow \mathsf{int}).\mathsf{fun}(g : (\mathsf{int} \rightarrow \mathsf{int}) \rightarrow \mathsf{int}).(f\ 1) \times (g\ f)$$

But is the most precise type really the best type? The answer depends on how the original function was used. For example, in the following program F_2 is not substitutable for F_1:

Before (produces zero)	After (static type error)
$\underline{F_1}\ (\mathsf{fun}(x : \star).0)\ (\mathsf{fun}(k : \mathsf{bool} \rightarrow \star).k\ \mathsf{true})$	$\underline{F_2}\ (\mathsf{fun}(x : \star).0)\ (\mathsf{fun}(k : \mathsf{bool} \rightarrow \star).k\ \mathsf{true})$

The left-hand side type-checks and evaluates to 0, while the right-hand side has a static type error: the bool type in the (unmigrated) context is inconsistent with the migrated type int.

We might reason that it is acceptable to generate this static error. But there is a second, more serious problem: in a gradually typed language, it is possible to turn static type errors into run-time type errors. Consider the following variation where the annotation on k in the unmigrated version is less precise:

Before (produces zero)	After (dynamic type error)
$\underline{F_1}\ (\mathsf{fun}(x : \star).0)\ (\mathsf{fun}(k : \star).k\ \mathsf{true})$	$\underline{F_2}\ (\mathsf{fun}(x : \star).0)\ (\mathsf{fun}(k : \star).k\ \mathsf{true})$

Both programs above are well-typed. However, the static error from the previous example is now a dynamic error. As we argued in §2, making types more precise in a portion of a program can introduce run-time errors at the (higher-order) boundary between migrated and unmigrated code.

Perhaps we can address this problem by producing a different migration of F_1:

$$F_3 \triangleq \mathsf{fun}(f : \star \rightarrow \mathsf{int}).\mathsf{fun}(g : (\star \rightarrow \mathsf{int}) \rightarrow \mathsf{int}).(f\ 1) \times (g\ f)$$

This migration is less precise than F_2: although f and g must still be functions, they are not required to consume integers. It is therefore equivalent to F_1 in our unmigrated context.

Before (produces zero)	After (also produces zero)
$\underline{F_1}\ (\mathsf{fun}(x : \star).0)\ (\mathsf{fun}(k : \star).k\ \mathsf{true})$	$\underline{F_3}\ (\mathsf{fun}(x : \star).0)\ (\mathsf{fun}(k : \star).k\ \mathsf{true})$

Unfortunately, there are other contexts that lead to errors in F_3 that do not occur with F_1. For instance, the following program produces an error with F_3 but not F_1.

$$F_3\ (\mathsf{fun}(x : \star).x)\ (\mathsf{fun}(id : \star).(\mathsf{fun}(b : \star).0)\ (id\ \mathsf{true}))$$

We can address this problem with a migration with even lower precision:

$$F_4 \triangleq \mathsf{fun}(f : \star \rightarrow \star).\mathsf{fun}(g : (\star \rightarrow \star) \rightarrow \mathsf{int}).(f\ 1) \times (g\ f)$$

This expression does not produce the same error as the previous example, and is compatible with all our examples. However, we have lost a lot of information about how F_1 uses its arguments. To summarize, we have seen a series of migrations for F_1 in decreasing order of precision:

$$F_1 \sqsubseteq F_4 \sqsubseteq F_3 \sqsubseteq F_2$$

Our algorithm produces F_2, but the other, less precise migrations are compatible with more contexts. So, which migration is best? The answer depends on the context of use for the program. If the programmer is generating documentation, they may prefer the more precise migration. On the other hand, if they are adding types to a library and cannot make assumptions about the function's caller, they may desire the migration that is compatible with more contexts.

```
1: ▷ The only annotations in $e$ are $\star$
2: function MIGRATE(WEAKEN, e)
3:     $e_1 \leftarrow$ INTRODUCEMETAVARS($e$)    ▷ Replace $\star$s with fresh $\alpha$s
4:     $\cdot \vdash e_1 \Rightarrow e', T_1, \phi$
5:     for $\alpha \in \phi$ do    ▷ The type metavariables in $\phi$
6:         (declare-const $\alpha$ Typ)
7:     for $w \in \phi$ do    ▷ The weight variables in $\phi$
8:         (declare-const $w$ Bool)
9:         (assert-soft $w$ 1)
10:    (check-sat $\phi$)
11:    $\sigma \leftarrow$ (get-model)
12:    $T_2 \leftarrow$ SUBST($\sigma, T_1$)    ▷ The most precise type
13:    $\phi' \leftarrow$ WEAKEN($T_2, T_1$)
14:    (check-sat $\phi \wedge \phi'$)
15:    $\sigma' \leftarrow$ (get-model)
16:    return SUBST($\sigma', e'$)
```

$$\boxed{T_1 = \star \text{ [Rastogi et al. 2012]}}$$

$$\begin{aligned}
P(T_1 \rightarrow T_2, \phi_T, b) &\triangleq \boxed{P(T_1, (\texttt{arr-in } \phi_T), \neg b)} \wedge \\
&\quad P(T_2, (\texttt{arr-out } \phi_T), b) \\
P(B, \phi_T, \text{true}) &\triangleq \text{true} \\
P(B, \phi_T, \text{false}) &\triangleq \phi_T = \star \\
P(\star, \phi_T, b) &\triangleq \text{true} \\
\textsc{Weaken}(T, \phi_T) &\triangleq P(T, \phi_T, \text{true})
\end{aligned}$$

Fig. 12. The Type Migration Algorithm.

4.4 Choosing Alternative Migrations

Although the algorithm presented above produces the most precise migration that the TYPEWHICH constraints encode, we can also use TYPEWHICH to infer alternative migrations that prioritize other properties, such as contextual compatibility.

At first glance, it seems straightforward to weaken the more precise type inferred in the preceding section. Suppose the algorithm produces a migration e with type T, and we want a less precise type S ($S \sqsubseteq T$). It seems that we could simply wrap e in a coercion: $[\mathit{coerce}(T, S)]e$. Unfortunately, this purported solution is no different from the adversarial contexts presented above. The expression has the desired weaker type S, but gradual typing ensures that it behaves the same as the stronger type T at run-time, including producing the same run-time errors! Instead, we need to alter the type annotations that are *internal* to e.

Weakening Migrations. TYPEWHICH employs a two-step approach to type migration. We first generate constraints and calculate the most precise type possible, as described earlier (lines 3–12 of Figure 12; identical to Figure 11). We then identify all base types in negative position (following Rastogi et al. [2012]), and add new constraints that force them to be $\star$ to ensure contextual compatibility. We apply a function WEAKEN to the output of the first-pass constraint generation (T_1, which has metavariables) to add these additional constraints.

Once we have the weakening constraint, we must update the type annotations in the migrated program and calculate the new weaker type. To do so, we run the solver once more with the added constraint (line 14). This produces a new model (line 15), which we use to substitute type metavariables and produce a fully annotated program.

It is worth reflecting on why a two-stage procedure is necessary. The first stage produces the most precise type that we can. This is necessary to discover a type skeleton that is as precise as possible; otherwise, we might miss some of the structure, e.g., by failing to predict arrow types. The second stage is necessary in order to propagate the constraints on the program's type back through the migrated program, which may involve arbitrary changes to internal type annotations.

Critically, the new set of constraints ϕ' must not impose unnecessary conditions on the type of the program. For example, suppose the original program e has a precise type int $\rightarrow$ int. Since this type only allows the context to provide int-arguments to e, we might conclude that a better type for e is $\star \rightarrow$ int. But this may be impossible: for instance, if e is the identity function, its argument

Ground types
$G ::= \cdots \mid \mathsf{ref}$

Base Types
$B ::= \cdots \mid \mathsf{unit}$

Constants
$b ::= \cdots \mid \mathsf{unit}$

Types
$T ::= \cdots \mid \mathsf{ref}\ T$

Expressions
$e ::= \cdots$

$\mid \mathsf{ref}\ e$	Create cell
$\mid\ !e$	Read cell
$\mid e_1 {:=} e_2$	Write cell

Constraint Metafunctions

$$\mathit{ground} \in T \rightarrow \phi$$
$$\mathit{ground}(T) = T \in B \vee T = \star \rightarrow \star \vee T = \mathsf{ref}\ \star$$

Type Representation

```
(declare-datatypes ()
  ((Typ (star) (int) (bool)
        (ref (to Typ))
        (arr (in Typ) (out Typ)))))
```

$$\textsc{If}\ \frac{\begin{array}{c}\Gamma \vdash e_1 \Rightarrow e_1', T_1, \phi_1 \qquad \Gamma \vdash e_2 \Rightarrow e_2', T_2, \phi_2 \qquad \Gamma \vdash e_3 \Rightarrow e_3', T_3, \phi_3 \qquad w_1, w_2, \alpha \text{ are fresh}\\ \phi_4 = ((T_1 = \mathsf{bool} \wedge w_1) \vee (T_1 = \star \wedge \neg w_1)) \wedge ((T_2 = T_3 = \alpha \wedge w_2) \vee (\alpha = \star \wedge \mathit{ground}(T_2) \wedge \mathit{ground}(T_3) \wedge \neg w_2))\end{array}}{\Gamma \vdash \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \Rightarrow \mathsf{if}\ [\underline{\mathit{coerce}}(T_1, \mathsf{bool})]e_1'\ \mathsf{then}\ [\underline{\mathit{coerce}}(T_2, \alpha)]e_2'\ \mathsf{else}\ [\underline{\mathit{coerce}}(T_3, \alpha)]e_3', \alpha, \phi_1 \wedge \phi_2 \wedge \phi_3 \wedge \phi_4}$$

$$\textsc{Add}\ \frac{\begin{array}{c}\Gamma \vdash e_1 \Rightarrow T_1, e_1' \phi_1 \qquad \Gamma \vdash e_2 \Rightarrow T_2, e_2', \phi_2 \qquad w, \alpha \text{ are fresh}\\ \phi_3 = ((\alpha = \mathsf{int} \vee \alpha = \mathsf{str}) \wedge \alpha = T_1 = T_2 \wedge w) \vee (\alpha = \star \wedge \mathit{ground}(T_1) \wedge \mathit{ground}(T_2) \wedge \neg w)\end{array}}{\Gamma \vdash e_1 + e_2 \Rightarrow ([\underline{\mathit{coerce}}(T_1, \alpha)]e_1') + ([\underline{\mathit{coerce}}(T_2, \alpha)]e_2'), \alpha, \phi_1 \wedge \phi_2 \wedge \phi_3}$$

$$\textsc{Ref}\ \frac{\begin{array}{c}\Gamma \vdash e \Rightarrow T, e', \phi_1 \qquad \alpha \text{ and } w \text{ are fresh}\\ \phi_2 = (\alpha = \mathsf{ref}\ T \wedge w) \vee (\alpha = \star \wedge \mathit{ground}(\mathsf{ref}\ T) \wedge \neg w)\end{array}}{\Gamma \vdash \mathsf{ref}\ e \Rightarrow [\underline{\mathit{coerce}}(\mathsf{ref}\ T, \mathsf{ref}\ \alpha)]\mathsf{ref}\ e', \alpha, \phi_1 \wedge \phi_2}$$

$$\textsc{Deref}\ \frac{\Gamma \vdash e \Rightarrow T, e', \phi_1 \quad \alpha, w \text{ are fresh} \quad \phi_2 = (T = \mathsf{ref}\ \alpha \wedge w) \vee (T = \alpha = \star \wedge \neg w)}{\Gamma \vdash !e \Rightarrow !([\underline{\mathit{coerce}}(T, \mathsf{ref}\ \alpha)]e'), \alpha, \phi_1 \wedge \phi_2}$$

$$\textsc{SetRef}\ \frac{\begin{array}{c}\Gamma \vdash e_1 \Rightarrow T_1, e_1', \phi_1 \qquad \Gamma \vdash e_2 \Rightarrow T_2, e_2', \phi_2 \qquad \alpha \text{ and } w \text{ are fresh}\\ \phi_3 = (T_1 = \mathsf{ref}\ \alpha \wedge T_2 = \alpha \wedge w) \vee (\alpha = \star \wedge \mathit{ground}(T_1) \wedge \mathit{ground}(T_2) \wedge \neg w)\end{array}}{\Gamma \vdash e_1 {:=} e_2 \Rightarrow ([\underline{\mathit{coerce}}(T_1, \mathsf{ref}\ \alpha)]e_1') {:=} [\mathit{coerce}(T_2, \alpha)]e_2', \mathsf{Unit}, \phi_1 \wedge \phi_2 \wedge \phi_3}$$

Fig. 13. Extensions to the GTLC.

and result types must be the same. On the other hand, if the body of e is a multiplication, then making the input type $\star$ does not affect the output type: it can remain int. By adding the constraint and re-solving, TypeWhich is able to distinguish between these two scenarios.

We note that there are several possible variations for Weaken. When migrating higher-order functions, it is useful to use a definition that turns base-typed inputs in negative position to $\star$, but preserves arrow types in the input. An alternative is to turn all input types to $\star$ to maximize compatibility, similar to Rastogi et al. [2012]. Our implementation of TypeWhich supports both of these and could be easily extended to other variations as well.

Our two-stage approach to contextual safety highlights the key trade-off between precision and safety in type migration. Our first-pass discovers the most precise types that we can; our second-pass sacrifices some of this precision to provide compatibility with a wider range of contexts.

Theorem 4.2 (Type Migration Completeness). *Every well scoped dynamic program e has a migration, i.e., there exists e', T, and ϕ such $\cdot \vdash e \Rightarrow e', T, \phi$ such that ϕ is satisfiable in some model σ.*

Proof. We prove that a fully dynamic model σ exists (Theorem D.1) and then show that such models are stable under weakening, i.e., they are still satisfiable (Lemma D.2 and Corollary D.3). □

5 LANGUAGE EXTENSIONS

We now extend the GTLC and TypeWhich to support several common language features. These new features affect our constraint generation rules, but they do not change the migration algorithm.

Conditionals. Retrofitted type checkers for untyped languages employ a variety of techniques to give precise types to conditional expressions (§7). The GTLC-based languages (e.g., Kuhlenschmidt et al. [2019]) use a simpler approach: (1) the type of the test must be consistent with bool, and (2) the type of the expression is the least upper bound of the types of either branch.

The If rule in Figure 13 shows constraint generation for conditionals. The generated constraint (ϕ_4) has two conjunctions that 1) constrain the type of the condition to bool or $\star$, and 2) constrain the types of each branch to be identical types or distinct ground types (in which case, both are coerced to the unknown type).

Overloaded Operators. Many languages have overloaded built-in operators: for instance, the "+" operator is frequently used for addition and string concatenation. To support this, the run-time system has three operators available: (1) primitive addition, (2) primitive string concatenation, and (3) a complex operation whose behavior depends on the run-time types of its arguments. Type migration can reveal the type at which an overloaded operator is used, which can help programmers understand their code and improve run-time performance. The constraint generation rule for "+" in Figure 13 introduces a boolean-sorted variable (w) that is true when the operands both have type int or str; when the variable is false, the constraint requires the two arguments to have type $\star$. Thus, it favors solutions that do not employ $\star$ when possible.

Mutable Data Structures. TypeWhich supports ML-style mutable references and mutable vectors. There are several ways to add mutable references to the GTLC [Herman et al. 2011; Siek and Taha 2006; Siek et al. 2015c]. However, all approaches share the following property: in untyped code, where all mutable cells contain $\star$-typed values, the *only* reason that reading or writing fails is when the expression in reference position is not a reference. In constraint generation, we are careful to avoid solutions that may introduce other kinds of errors.

The least precise reference type is a reference to the unknown type (ref $\star$), so we add this to the set of ground types (Figure 13). In the constraint generation rule for writes, we require that either (1) the type of value written is exactly the referenced type, or (2) both the reference and the value written are ground types. The restriction to ground types is necessary because, as in the function case, once a reference is coerced to $\star$, we have no way to recover its original type; allowing non-ground types to be coerced to $\star$ can introduce run-time errors. TypeWhich also supports mutable vectors implemented along the same lines.

Other language features. The implementation of TypeWhich supports a variety of other language features, including tuples, `let`, and a `fix` construct. Many of these are necessary to support the Grift programming language, which we use in our evaluation. Constraint generation rules for these extensions can be found in Appendix A.

6 EVALUATION

This section presents the first comprehensive comparison of several type migration algorithms from the literature (along with TypeWhich). We compare five type migration tools on a two-part suite of 22 programs: the benchmarks from Migeed and Palsberg [2020] and a new suite of "challenge programs" that we have designed to illustrate the strengths and weaknesses of various approaches. We also evaluate TypeWhich using the Grift benchmarks from Kuhlenschmidt et al. [2019], to show that TypeWhich can reconstruct hand-written type annotations in Grift.

To facilitate high-level understanding of the results, we first discuss summary metrics for evaluating automated type migration tools.

6.1 How Should Type Migration Tools Be Evaluated?

As we have argued in §2, the type migration problem involves inherent trade-offs between different goals. For this reason, we avoid choosing a single evaluation metric, since this would favor one goal of type migration over the others. For instance, using the total number of type annotations improved is a good metric for type precision, but reporting only precision obscures the fact that not all type refinements are alike: some change the behavior of the original program, while others preserve its semantics. We have also illustrated how type precision can come at the expense of compatibility with unmigrated code. This sacrifice may sometimes be warranted, but when a function is migrated, it should remain usable with at least *some arguments*. This seems like a trivial point, but consider the following migration:

Original Program	Migrated Program
$\mathsf{fun}(f : \star).\mathsf{fun}(x : \star).f x x$	$\mathsf{fun}(f : \mathsf{int} \rightarrow \mathsf{bool} \rightarrow \star).\mathsf{fun}(x : \star).f x x$

The migrated program has types that cannot be made more precise. However, the type of f requires x to be both an integer and a boolean, and thus renders the function unusable.

We propose a multi-stage evaluation process for automated type migration tools. For each tool, (1) we start with the full suite of programs and ask, *How many programs does the tool reject with static errors?* (2) We take the *remaining programs* and ask, *How many migrated programs crash with a new dynamic type error?* (3) We take the *remaining programs* and ask, *How many migrated programs are functions that are rendered unusable?* (4) We take the remaining programs and ask two final questions: (a) *How many migrated programs are functions with types that are incompatible with some untyped contexts?* and (b) *How many type annotations, counted across all remaining programs, are not improved by migration?*

Note that the denominator (potentially) decreases at each stage: if a tool fails to migrate a program, then it is impossible to assess whether the migrated program crashes with a dynamic error. Moreover, we do not want to give a system credit for increasing the precision of a type if the refinement triggers a new dynamic error (i.e., it was an unsafe migration).

6.2 Type Migration Systems

We evaluate the performance of the following tools:

(1) TypeWhich: our tool, which we run in two modes: (a) to produce the most precise migration that we can (TypeWhich-P), and (b) to produce a migration that is compatible with unmigrated code (TypeWhich-C).
(2) Gtubi: *gradual typing with unification-based inference* [Siek and Vachharajani 2008] is the earliest work on gradual type migration. It does not introduce coercions that may fail.
(3) InsAndOuts: our implementation of the algorithm in Rastogi et al. [2012]. The algorithm is designed to not introduce coercions that may fail, and to produce a migration that is compatible with arbitrary unmigrated code.
(4) MaxMigrate: Migeed and Palsberg [2020] presents algorithms for several migration problems. We use the *maximal migration* tool, which produces a migration that cannot be made more precise. The tool searches for migrations by building types up to some depth (we use depth five as in the paper). A single program may have several maximal migrations; we take the first migration the tool produces. We halt with no output if no migration is found.
(5) MGT: our implementation of the algorithm in Campora et al. [2018b] for migrating untyped or partially typed programs. We start from untyped code (all functions annotated with $\star$), and take the first migration it produces.

Name	Expression
FArg-Mismatch	$(\mathsf{fun}(f:\star).f\ \mathsf{true})\ (\mathsf{fun}(x:\star).x+1)$
Rank2-Poly-Id	$(\mathsf{fun}(i:\star).(\mathsf{fun}(a:\star).(i\ \mathsf{true}))\ (i\ 5))\ (\mathsf{fun}(x:\star).x)$
Unreachable-Err	$(\mathsf{fun}(b:\star).b\ (\mathsf{fun}(c:\star).5\ 5)\ (\mathsf{fun}(d:\star).0))\ (\mathsf{fun}(t:\star).\mathsf{fun}(f:\star).f)$
F-In-F-Out*	$(\mathsf{fun}(f:\star).(\mathsf{fun}(y:\star).f)\ (f\ 5))\ (\mathsf{fun}(x:\star).10+x)$
Order3-Fun*	$\mathsf{fun}(f:\star).\mathsf{fun}(x:\star).x\ (f\ x)$
Order3-IntFun*	$\mathsf{fun}(f:\star).\mathsf{fun}(g:\star).f\ g\ ((g\ 10)+1)$
Double-F*	$\mathsf{fun}(f:\star).f\ (f\ \mathsf{true})$
Outflows*	$(\mathsf{fun}(x:\star).x\ 5+x)\ 5$
Precision-Relation*	$(\mathsf{fun}(f:\star).f\ \mathsf{true}+(\mathsf{fun}(g:\star).g\ 5)\ f)\ (\mathsf{fun}(x:\star).5)$
If-Tag	$\mathsf{fun}(tag:\star).\mathsf{fun}(x:\star).\mathsf{if}\ tag\ \mathsf{then}\ x+1\ \mathsf{else\ if}\ x\ \mathsf{then}\ 1\ \mathsf{else}\ 0$

Fig. 14. Our Type Migration Challenge.

6.3 Gradual Type Migration Benchmarks

We evaluate type migration tools using a two-part benchmark suite: a suite of benchmarks from Migeed and Palsberg [2020], and a new suite of *challenge programs* designed to illustrate the strengths and weaknesses of different approaches to type migration.

Our proposed challenge suite is presented in Figure 14. We describe the ten programs below. Although TypeWhich supports several extensions to the GTLC (§5), we largely avoid their use in the challenge suite for compatibility with as many approaches as possible.

1. FArg-Mismatch: crashes at run-time, because the functional argument f expects an integer, but is applied to a boolean.
2. Rank2-Poly-Id (based on Figure 5): defines the identity function and applies it to a number and a boolean. It uses a Church encoding of let-binding and sequencing that would require rank-2 polymorphism in an ML dialect.
3. Unreachable-Err (based on Figure 2): has a crashing expression similar to FArg-Mismatch, but it is unreachable. The example encodes a conditional as a Church boolean.
4. F-In-F-Out: defines a local function f that escapes.
5. Order3-Fun: a higher-order function that receives two functions f and x. Moreover, the body calculates $f\ x$, so f must be a higher-order function itself.
6. Order3-IntFun: similar to Order3-Fun, but the program uses operations that force several types to be int.
7. Double-F: calculates $f\ (f\ \mathsf{true})$. The inner application suggests that f's argument must be bool. However, that would rule out $\mathsf{fun}(x:\star).0$ as a possible value for f.
8. Outflows: defines a function that uses its argument as two different types. However, the function receives an integer.
9. Precision-Relation: names a function f that must receive $\star$, since f is applied twice to two different types. However, the second application re-binds f to g, thus g may have a more precise type.
10. If-Tag: receives a boolean and uses its value to determine the type of x. Conditionals are not in the core GTLC and not supported by all the tools that we consider. However, it is essential to think through conditionals, since they induce a type constraint between both branches, and a Church encoding incurs a significant loss of precision.

Some of these programs (marked with an asterisk in Figure 14) can be given types using Hindley-Milner type inference via translation into OCaml or Haskell. Doing so reveals important differences between conventional static types and the GTLC. For example, the most general type of Order3-Fun is a type scheme with two type variables. The GTLC does not support polymorphism, so a

	The tool rejects the program, e.g., $1 + \mathsf{true}$				
		$(\mathsf{fun}(id : \star).id\ 1)$ $(\mathsf{fun}(x : \mathsf{bool}).x)$ crashes			
			$(\mathsf{fun}(f : \mathsf{int} \rightarrow \mathsf{bool} \rightarrow \star).\mathsf{fun}(x : \star).f\,x\,x$ is unusable		
				$\mathsf{fun}(x : \mathsf{int}).x$ is restricted	
Tool	Rejected / Total Programs	New Dynamic Errors / Remaining Programs	Unusable Functions / Remaining Programs	Restricted Functions / Remaining Programs	Not Improved / Total ★
Gtubi	14 / 22	0 / 8	0 / 8	1 / 8	4 / 17
InsAndOuts	2 / 22	0 / 20	0 / 20	0 / 20	24 / 42
MGT	0 / 22	0 / 22	0 / 22	3 / 22	29 / 58
MaxMigrate	5 / 22	3 / 17	3 / 14	3 / 11	6 / 18
TypeWhich-C	0 / 22	0 / 22	0 / 22	0 / 22	39 / 58
TypeWhich-P	0 / 22	0 / 22	0 / 22	4 / 22	25 / 58

Fig. 15. Summary of type migration results. TypeWhich-C favors compatibility with unmigrated code, and TypeWhich-P favors precision. Above each column, we show an example of the kind of migrated program we count in that column.

type migration must use $\star$ rather than the more precise type. In contrast, the type of f in Double-F is $\mathsf{bool} \rightarrow \mathsf{bool}$. However, f can have other types in the GTLC.

6.4 Evaluation Results

The results of our evaluation illustrate the various strengths and weaknesses of different approaches to automated type migration. Before diving into the details of the complete results, we present a bird's-eye view using the evaluation scheme proposed in §6.1.

Figure 15 summarizes each tool's performance on the full benchmark suite. We include results from running TypeWhich in two different modes: TypeWhich-P prioritizes precision, while TypeWhich-C prioritizes contextual compatibility. By design, TypeWhich does not produce static or dynamic errors. When it is configured for type precision (TypeWhich-P), it does restrict the inputs of four functions. However, even in this mode, the remaining 18 programs remain compatible with all callers. On the other hand, when it is configured to prioritize contextual compatibility (TypeWhich-C), no programs are restricted, but fewer types are improved.

Like TypeWhich, MGT restricts some functions, but does not produce static or dynamic errors. Gtubi rejects several programs statically and restricts the behavior of some functions. However, it does not introduce any dynamic errors. MaxMigrate rejects a few programs: some do not have maximal migrations, on others it cannot find a migration within its search space, and one of our programs uses a conditional, which is unsupported. In addition, the tool introduces run-time errors in some programs, and makes some functions unusable. InsAndOuts rejects two programs.[4] On the remaining programs, it produces migrations that are compatible with arbitrary unmigrated code as intended. In fact, when we prioritize compatibility with unmigrated code, InsAndOuts outperforms all other approaches.

The right-most column of the table reports the number of type annotations that are *not* improved, and this must be interpreted very carefully. The point of gradual typing is that $\star$ serves as an "escape hatch" for programs that cannot be given more precise types. Our suite includes programs that *must* have some $\star$s, so every tool will have to leave some $\star$s unchanged. We naturally want a tool to improve as many types as possible, so we may prefer a tool that has the fewest number

[4]These are two programs from the Migeed and Palsberg [2020] benchmarks. From correspondence with the authors of Rastogi et al. [2012], our implementation seems faithful to the presentation in the paper, and the original implementation for Adobe ActionScript is no longer accessible.

of unimproved types. However, notice that the denominator varies considerably. For example, TypeWhich-P cannot improve about half the annotations, but it does not introduce any errors. In contrast, the oldest tool—Gtubi–only leaves a small fraction of annotations unimproved, but it statically rejects the majority of programs.

6.4.1 Challenge Set Results. We now examine performance on the challenge set in more detail. Figure 16 shows the migrated challenge programs produced by these tools. We present and discuss results produced by running TypeWhich to prioritize precision (TypeWhich-P); results from TypeWhich-C can be found in the appendix.

Examining the detailed output on the challenge set programs reveals interesting differences in the migrations inferred by the various type migration tools, reflecting their differing priorities.

1. FArg-Mismatch: InsAndOuts and MaxMigrate produce the most precise and informative result: they show that x is a boolean next to $x + 100$, which helps locate the error in the program.
2. Rank2-Poly-Id: InsAndOuts and TypeWhich produce the best result that does not introduce a run-time error. MaxMigrate produces the most precise static type, but has a dynamic type error.
3. Unreachable-Err: TypeWhich, MGT, and InsAndOuts are the only tools that produce a result. The erroneous and unreachable portion gets the type $\star$ in TypeWhich; whereas InsAndOuts produces a type variable. The rest of the program has informative types.
4. F-In-F-Out: MGT, Gtubi, and TypeWhich produce the most precise result. MaxMigrate produces an alternative, equally precise type, but introduces a dynamic type error.
5. Order3-Fun: Gtubi produces the best result. Its result has type variables, thus is a type scheme. However, in a larger context, these variables would unify with concrete GTLC types. TypeWhich produces a needless int annotation that restricts the program. MaxMigrate produces $\mathsf{int} \rightarrow \mathsf{int}$ as the type of f, which is maximal, but introduces a subtle problem: $(f\ x)$ requires x to be an integer, but $x\ (f\ x)$ requires x to be a function.
6. Order3-IntFun: the results are similar to Order3-Fun, with Gtubi again doing the best. However, since the program forces certain types to be int, TypeWhich and MGT now produce the same result.
7. Double-F: MaxMigrate produces the best result. The most informative annotation on f that is compatible with all contexts is $\star \rightarrow \star$; no tool produces this type.
8. Outflows: InsAndOuts produces the best result. This program requires x to have two different types and thus crashes. Because the function receives an integer for x, Rastogi et al. [2012] gives x the type int. The other tools are not capable of reasoning in this manner. In a modification of this example where x is used with different types in each branch of a conditional, all tools would likely produce similar results.
9. Precision-Relation: InsAndOuts produces the most precise type that does not introduce a run-time type inconsistency. TypeWhich does not give g the most precise type; MGT does not improve the type of f; and MaxMigrate finds a maximal migration that constraints f's argument to bool.
10. If-Tag: Gtubi and MaxMigrate do not support conditionals. TypeWhich-P and MGT produce an unusual result that restricts the type of the argument x to bool and turns the $x + 1$ into $([\mathsf{int}?]x) + 1$. If we were migrating a larger program that had this function as a sub-expression, and this function were actually applied to different x arguments with different types, it would be $\star$.

FArg-Mismatch	MGT	$(\text{fun } f : \star . f \text{ true})(\text{fun } x : \text{int}. x + 100)$
	MaxMigrate	identical to InsAndOuts
	InsAndOuts	$(\text{fun } f : \text{bool} \to \text{int}. f \text{ true})(\text{fun } x : \text{bool}. x + 100)$
	TypeWhich-P	$(\text{fun } f : \star \to \text{int}. f \text{ true})(\text{fun } x : \star . x + 100)$
	Gtubi	constraint solving error
Rank2-Poly-Id	MGT	no improvement
	MaxMigrate	$(\text{fun } i : \star \to \star . (\text{fun } a : \text{int}. i \text{ true})(i 5))(\text{fun } x : \text{bool}. x)$
	InsAndOuts	identical to TypeWhich
	TypeWhich-P	$(\text{fun } i : \star \to \star . (\text{fun } a : \star . i \text{ true})(i\ 5))(\text{fun } x : \star . x)$
	Gtubi	constraint solving error
Unreachable-Err	MGT	less precise than TypeWhich-P
	MaxMigrate	No maximal migration
	InsAndOuts	$(\text{fun } b : (\alpha \to \alpha) \to (\star \to \text{int}) \to \star \to \text{int}. b (\text{fun } c : \star . 5\ 5)(\text{fun } d : \star . 0))$ $(\text{fun } t : \alpha \to \alpha . \text{fun } f : \star \to \text{int}. f)$
	TypeWhich-P	$(\text{fun } b : (\text{int} \to \star) \to (\text{int} \to \text{int}) \to \text{int} \to \text{int}. b (\text{fun } c : \text{int}.$ $([\text{int}!]5)\ 5)(\text{fun } d : \text{int}. 0))(\text{fun } t : \text{int} \to \star . \text{fun } f : \text{int} \to \text{int}. f)$
	Gtubi	constraint solving error
F-In-F-Out	MGT	identical to TypeWhich
	MaxMigrate	$(\text{fun } f : \text{int} \to \star . (\text{fun } y : \text{bool}. f)(f\ 5))(\text{fun } x : \text{int}. 10 + x)$
	InsAndOuts	$(\text{fun } f : \star \to \text{int}. (\text{fun } y : \text{int}. f)(f 5))(\text{fun } x : \star . 10 + x)$
	TypeWhich-P	$(\text{fun } f : \text{int} \to \text{int}. (\text{fun } y : \text{int}. f)(f 5))(\text{fun } x : \text{int}. 10 + x)$
	Gtubi	identical to TypeWhich
Order3-Fun	MGT	$\text{fun } f : (\star \to \star) \to \star . \text{fun } x : \star \to \star . x (f x)$
	MaxMigrate	$\text{fun} f : \text{int} \to \text{int}. \text{fun} x : \star . x (f x)$
	InsAndOuts	no improvement
	TypeWhich-P	$\text{fun } f : (\star \to \text{int}) \to \star . \text{fun } x : \star \to \text{int}. x (f x)$
	Gtubi	$\text{fun } f : (\alpha \to \beta) \to \alpha . \text{fun } x : \alpha \to \beta . x (f x)$
Order3-IntFun	MGT	identical to TypeWhich
	MaxMigrate	$\text{fun} f : \text{int} \to \text{int} \to \text{int}. \text{fun } g : \star . f g (g 10 + 1)$
	InsAndOuts	no improvement
	TypeWhich-P	$\text{fun } f : (\text{int} \to \text{int}) \to \text{int} \to \star . \text{fun } g : \text{int} \to \text{int}. f g (g 10 + 1)$
	Gtubi	$\text{fun } f : (\text{int} \to \text{int}) \to \text{int} \to \alpha . \text{fun } g : \text{int} \to \text{int}. f g (g 10 + 1)$
Double-F	MGT	identical to TypeWhich
	MaxMigrate	$\text{fun } f : \star \to \text{int}. f (f \text{True})$
	InsAndOuts	no improvement
	TypeWhich-P	$\text{fun } f : \text{bool} \to \text{bool}. f (f \text{true})$
	Gtubi	identical to TypeWhich
Outflows	MGT	no improvement
	MaxMigrate	$(\text{fun} x : \star . x\ 5 + x)\ 5$
	InsAndOuts	$(\text{fun} x : \text{int}. x\ 5 + x)\ 5$
	TypeWhich-P	identical to MaxMigrate
	Gtubi	constraint solving error
Precision-Relation	MGT	$(\text{fun} f : \star . (f \text{ true}) + ((\text{fun} g : \text{int} \to \star . g\ 5) f))(\text{fun} x : \star . 5)$
	MaxMigrate	$(\text{fun } f : \text{bool} \to \text{int}. (f \text{true}) + ((\text{fun } g : \star \to \text{int}. g 5) f))(\text{fun } x : \text{bool}. 5)$
	InsAndOuts	$(\text{fun } f : \star \to \text{int}. (f \text{true}) + ((\text{fun } g : \text{int} \to \text{int}. g 5) f))(\text{fun } x : \star . 5)$
	TypeWhich-P	$(\text{fun } f : \star \to \text{int}. (f \text{true}) + ((\text{fun } g : \star \to \text{int}. g 5) f))(\text{fun } x : \star . 5)$
	Gtubi	constraint solving error
If-Tag	MGT	Identical to TypeWhich-P
	MaxMigrate	conditionals unsupported
	InsAndOuts	$\text{fun} tag : \star . \text{fun} x : \star . \text{if} tag \text{then} x + 1 \text{elseif } x \text{ then } 1 \text{ else } 0$
	TypeWhich-P	$\text{fun } tag : \text{bool}. \text{fun } x : \text{bool}. \text{if } tag \text{ then } ([\text{int}!] x) + 1 \text{ else if } x \text{ then } 1 \text{ else } 0$
	Gtubi	conditionals unsupported

Fig. 16. Migrations of the challenge set with TypeWhich in precise mode.

6.4.2 Migeed and Palsberg [2020] Benchmarks. Migeed and Palsberg [2020] compare their maximal migration tool to the type migration tool in Campora et al. [2018b]. We extend the comparison to include TypeWhich, Gtubi, and InsAndOuts. The complete results are in Appendix E, and we include all of these benchmarks in our summary (Figure 15).

6.4.3 Summary. Our type migration challenge suite is designed to highlight the strengths and weaknesses of different algorithms. As discussed in §2, the competing goals of the type migration problem lead to a range of compromises; we do not claim that any one approach is best, since each approach reflects a different weighting of priorities. Because our challenge programs are synthetic, it would be possible to build a large set of programs that favor one tool at the expense of others. Our goal has been instead to curate a small set that illustrates a variety of weaknesses in every tool. In addition, our challenge programs are unlikely to be representative of real-world type migration problems. A more thorough evaluation would require scaling type migration tools to a widely-used language with a corpus of third-party code, which is beyond the scope of this paper.

6.5 Grift Performance Benchmarks

Kuhlenschmidt et al. [2019] present a benchmark suite to evaluate the performance of Grift programs (running time and space efficiency). Grift extends the GTLC with floating-point numbers, characters, loops, recursive functions, tuples, mutable references, vectors, and several primitive operators. Each benchmark has two versions: an untyped version and a fully-typed, hand-annotated version. We use TypeWhich (in precise mode) to migrate every untyped benchmark, and compare the result to the human type annotations. TypeWhich supports all Grift features except equirecursive types. However, because Grift's equirecursive types do not introduce new expression forms, TypeWhich can still be run on all programs: it just fails to improve annotations that require them.

TypeWhich performs as follows on the Grift benchmarks:

- **On 9 of 11 benchmarks**, TypeWhich produces exactly the same type annotations as the hand-typed version.
- **N-body** defines a number of unused functions over vectors. Since they are under-constrained, TypeWhich makes some arbitrary choices. On the reachable portion of the benchmark, we produce exactly the same type annotations as the hand-typed version.
- **Sieve** defines a stream library, and the hand-typed version gives streams an equirecursive type. TypeWhich improves some types, but it cannot improve the annotations on the stream library. TypeWhich infers $\star$ rather than the `Tuple Int Dyn` migration shown below:

  ```
  (define (stream-rest [st : (Tuple Int Dyn)])
    : (Tuple Int Dyn)
    ((tuple-proj st 1)))
  ```

  The projection from the stream has type $\star$, but the function expects to return a tuple. However, our constraints will only insert a coercion from $\star$ to a more precise type at an elimination form, so TypeWhich will not produce this migration.

6.6 Implementation and Performance

The TypeWhich tool is open-source and written in approximately 12,000 lines of Rust. This code includes our new migration algorithm, implementation of the migration algorithm from Rastogi et al. [2012] and Campora et al. [2018b], and a unified evaluation framework that supports all the third-party tools that we use in our evaluation. The evaluation framework is designed to automatically validate the evaluation results we report. For example, to report that a migrated function is not compatible with all untyped contexts, our framework requires an example of a context that distinguishes between the migrated and original program, and runs both programs in the given context to verify that they differ. The framework also ensures that migrated programs are well-typed and structurally identical to the original program.

We perform all our experiments on on a virtual machine with 4 CPUs and 8 GB RAM, running on an AMD EPYC 7282 processor. The full suite consists of 892 LOC and 33 programs. TypeWhich produces migrations for our entire suite of benchmarks in under three seconds.

7 RELATED WORK

There is a growing body of work on automating gradual type migration and related issues. Our work is most closely related to the four algorithms we evaluate in §6. Siek and Vachharajani [2008] substitutes metavariables that appear in type annotations with concrete types, using a variation on unification. Rastogi et al. [2012] builds a type inference system for ActionScript. Their system ensures that inference never fails and produces types that are compatible with all untyped contexts. Campora et al. [2018b] uses variational typing to heuristically tame the exponential search space of types [Chen et al. 2014]. Migeed and Palsberg [2020] present decidability results for several type migration problems, including finding maximally precise migrations.

The aforementioned work relies on custom constraint solving algorithms. A key contribution of this paper is that sets up gradual type migration for an off-the-shelf MaxSMT solver, which makes it easier to build a type migration tool. In addition, we present a comprehensive evaluation comparing all five approaches. As part of this effort, we have produced new, open-source implementations of the algorithms presented in Rastogi et al. [2012] and Campora et al. [2018b].

Henglein [1994] introduces the theory of coercions that we use; Henglein and Rehof [1995] present an efficient compiler from Scheme to ML that inserts coercions when necessary. This work also uses a custom constraint solver and a complex graph algorithm. The latter defines a *polymorphic safety* criterion, which is related to our notion of a context-restricted type migration (Definition 3.4).

Garcia and Cimini [2015] extend Siek and Vachharajani's work to infer principal types. Since we focus on monomorphic types, we do not directly compare against their algorithm. Miyazaki et al. [2019] build on Garcia and Cimini's work, discussing the coherence issues what we point out in §2: types induce run-time checks that can affect program behavior. However, while we migrate all programs, Miyazaki et al. use *dynamic type inference* to discover type inconsistencies and report them as run-time errors. Castagna et al. [2019] propose another account of gradual type inference that supports many features (let-polymorphism, recursion, and set-theoretic types). They do not consider run-time safety. Finally, Campora et al. [2018a] extend their previous work [2018b] with a cost model for selecting migrations. Like us, they discuss trade-offs in type migration, although they focus on type precision and performance, rather than semantics preservation.

Tobin Hochstadt and Felleisen [2008], Guha et al. [2011], Chugh et al. [2012], and Vekris et al. [2015] are examples of retrofitted type checkers for untyped languages that feature flow-sensitivity. These tools require programmers to manually migrate their code, while we focus on automatic type migration. However, they go beyond our work by considering flow-sensitivity.

Anderson et al. [2005] present type inference for a representative fragment of JavaScript. However, the approach is not designed for gradual typing, where portions of the program may be untyped. Similarly, Chandra et al. [2016] infer types for JavaScript programs with the goal of compiling them to run efficiently on low-powered devices; their approach is not gradual by design and deliberately rejects certain programs.

Pavlinovic et al. [2014] formulate a MaxSMT problem to localize OCaml type errors. We also use MaxSMT and encode types in a similar manner. However, both the form of our constraints and the role of the MaxSMT solver are very different. In error localization, the MaxSMT problem helps isolate type errors from well-typed portions of the program. In our work, the entire program must be well-typed. Moreover, our constraints allow several typings, and we use soft constraints to guide the MaxSMT solver towards solutions with fewer coercions.

Soft Scheme [Wright and Cartwright 1997] infers types for Scheme programs. However, its type system is significantly different from the GTLC, which hinders comparisons to contemporary type migration tools for the GTLC. Flanagan [1997, p. 41]'s discussion of how Soft Scheme's sophistication can lead to un-intuitive types inspired work on set-based analysis of Scheme programs: Flanagan et al. [1996] map program points to sets of abstract values, rather than types.

Alternative approaches to type migration and type inference consider sources of evidence beyond the program to be migrated. This includes work that applies supervised machine learning techniques to generate type annotations, such as Hellendoorn et al. [2018]; Malik et al. [2019]; Pradel et al. [2020]; Wei et al. [2020]. Other lines of work use run-time profiling to guide type inference [An et al. 2011; Furr et al. 2009; Saftoiu 2010], or use programmer-supplied heuristics to guide type inference [Kazerounian et al. 2020; Ren and Foster 2016].

8 CONCLUSION

We present TypeWhich, a new approach to type migration for the GTLC that is more flexible than previous approaches in two key ways. First, we formulate constraints for an off-the-shelf MaxSMT solver rather than building a custom constraint solver, which makes it easier to extend TypeWhich. We demonstrate this flexibility by adding support for several language features beyond the core GTLC. Second, TypeWhich can produce alternative migrations that prioritize different goals, such as type precision and compatibility with unmigrated code. This makes TypeWhich a more flexible approach, suitable for migration in multiple contexts.

We also contribute to the evaluation of type migration algorithms. We define a multi-stage evaluation process that accounts for multiple goals of type migration. We present a "type migration challenge set": a benchmark suite designed to illustrate the strengths and weaknesses of various type migration algorithms. We evaluate TypeWhich alongside four existing type migration systems. Toward this end, we contribute open-source implementations of two existing algorithms from the literature, which we incorporate into a unified framework for automated type migration evaluation. We hope these evaluation metrics, new benchmarks, and benchmarking framework will aid future work by illuminating the differences among the many approaches to gradual type migration.

REFERENCES

Jong-hoon David An, Avik Chauduri, Jeffrey S. Foster, and Michael Hicks. 2011. Dynamic Inference of Static Types for Ruby. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*.

Christopher Anderson, Paola Giannini, and Sophia Drossopoulou. 2005. Towards Type Inference for JavaScript. In *European Conference on Object-Oriented Programming (ECOOP)*.

Clark Barrett, Iger Shikanian, and Cesare Tinelli. 2007. An Abstract Decision Procedure for a Theory of Inductive Data Types. *Journal on Satisfiability, Boolean Modeling and Computation* 3, 1–2 (July 2007), 21–46.

Nikolaj Bjørner, Anh-Dung Phan, and Lars Fleckenstein. 2015. *ν*Z: An Optimizing SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*.

John Peter Campora, Sheng Chen, Martin Erwig, and Eric Walkingshaw. 2018b. Migrating Gradual Types. *Proceedings of the ACM on Programming Languages (PACMPL)* 2, POPL, Article 15 (Dec. 2018), 29 pages.

John Peter Campora, Sheng Chen, and Eric Walkingshaw. 2018a. Casts and Costs: Harmonizing Safety and Performance in Gradual Typing. *Proceedings of the ACM on Programming Languages (PACMPL)* 2, ICFP (2018).

Giuseppe Castagna, Victor Lanvin, Tommaso Petrucciani, and Jeremy G. Siek. 2019. Gradual Typing: A New Perspective. *Proceedings of the ACM on Programming Languages (PACMPL)* 3, POPL (2019).

Satish Chandra, Colin S. Gordon, Jean-Baptiste Jeannin, Cole Schlesinger, Manu Sridharan, Frank Tip, and Young-Il Choi. 2016. Type inference for static compilation of JavaScript. In *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA)*.

Sheng Chen, Martin Erwig, and Eric Walkingshaw. 2014. Extending Type Inference to Variational Programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 36, 1 (2014).

Ravi Chugh, Patrick M. Rondon, and Ranjit Jhala. 2012. Nested Refinements for Dynamic Languages. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*.

P. Dubey. 2005. Recognition, Mining and Synthesis Moves Computers to the Era of Tera.

Cormac Flanagan. 1997. *Effective Static Debugging via Componential Set-based Analysis*. Ph.D. Dissertation. Rice University.

Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, Stephanie Weirich, and Matthias Felleisen. 1996. Catching Bugs in the Web of Program Invariants. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

Michael Furr, Jong-hoon David An, and Jeffrey S. Foster. 2009. Profile-Guilding Static Typing for Dynamic Scripting Languages. In *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA)*.

Richard P. Gabriel. 1985. *Performance and Evaluation of LISP Systems*. Massachusetts Institute of Technology, USA.

Ronald Garcia and Matteo Cimini. 2015. Principal Type Schemes for Gradual Programs. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*.

Paul Graham. 1995. *ANSI Common Lisp*. Prentice Hall.

Michael Greenberg. 2013. *Manifest Contracts*. Ph.D. Dissertation. University of Pennsylvania.

Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. 2011. Typing Local Control and State Using Flow Analysis. In *European Symposium on Programming (ESOP)*.

Vincent J. Hellendoorn, Christian Bird, Earl T. Barr, and Miltiadis Allamanis. 2018. Deep Learning Type Inference. In *ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*.

Fritz Henglein. 1994. Dynamic typing: syntax and proof theory. *Science of Computer Programming* 22, 3 (1994), 197–230.

Fritz Henglein and Jakob Rehof. 1995. Safe polymorphic type inference for a dynamically typed language: Translating Scheme to ML. In *International Conference on functional programming languages and computer architecture (FPCA)*.

David Herman, Aaron Tomb, and Cormac Flanagan. 2011. Space-efficient gradual typing. *Higher-Order and Symbolic Computation (HOSC)* 23, 2 (2011), 167–189.

Milod Kazerounian, Brianna M. Ren, and Jeffrey S. Foster. 2020. Sound, Heuristic Type Annotation Inference for Ruby. In *Dynamic Languages Symposium (DLS)*.

Brian W. Kernighan and Christopher J. Van Wyk. 1998. Timing trials, or the trials of timing: experiments with scripting and user-interface languages. *Software: Practice and Experience* 28, 8 (1998), 819–843.

Andre Kuhlenschmidt, Deyaaeldeen Almahallawi, and Jeremy G. Siek. 2019. Toward Efficient Gradual Typing for Structural Types via Coercions. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

Rabee Sohail Malik, Jibesh Patra, and Michael Pradel. 2019. NL2Type: Inferring JavaScript Function Types from Natural Language Information. In *Proceedings of the 41st International Conference on Software Engineering* (Montreal, Quebec, Canada) *(ICSE '19)*. IEEE Press, 304–315.

Jacob Matthews and Robert Bruce Findler. 2009. Operational Semantics for Multi-Language Programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 31, 3 (2009).

Zeina Migeed and Jens Palsberg. 2020. What is Decidable about Gradual Types? *Proceedings of the ACM on Programming Languages (PACMPL)* 4, POPL, Article 29 (Dec. 2020), 29 pages.

Yusuke Miyazaki, Taro Sekiyama, and Atsushi Igarashi. 2019. Dynamic Type Inference for Gradual Hindley–Milner Typing. *Proceedings of the ACM on Programming Languages (PACMPL)* 3, POPL (2019).

Zvonimir Pavlinovic, Tim King, and Thomas Wies. 2014. Finding Minimum Type Error Sources. In *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA)*.

Michael Pradel, Georgios Gousios, Jason Liu, and Satish Chandra. 2020. TypeWriter: Neural Type Prediction with Search-Based Validation. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Virtual Event, USA) *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 209–220.

Aseem Rastogi, Avik Chaudhuri, and Basil Hosmer. 2012. The Ins and Outs of Gradual Type Inference. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*.

Brianna M. Ren and Jeffrey S. Foster. 2016. Just-in-Time Static Type Checking for Dynamic Languages. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

Claudiu Saftoiu. 2010. *JSTrace: Run-time type discovery for JavaScript*. Master's thesis. Brown University.

Jeremy Siek, Peter Thiemann, and Philip Wadler. 2015a. Blame and Coercion: Together Again for the First Time. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

Jeremy Siek, Michael Vitousek, Matteo Cimini, and John Boyland. 2015b. Refined Criteria for Gradual Typing. In *Summit oN Advances in Programming Languages (SNAPL)*.

Jeremy G. Siek and Walid Taha. 2006. Gradual Typing for Functional Languages. In *Scheme and Functional Programming Workshop (SW)*.

Jeremy G. Siek and Manish Vachharajani. 2008. Gradual Typing with Unification-based Inference. In *Dynamic Languages Symposium (DLS)*.

Jeremy G. Siek, Michael M. Vitousek, Matteo Cimini, Sam Tobin-Hochstadt, and Ronald Garcia. 2015c. Monotonic References for Efficient Gradual Typing. In *European Symposium on Programming (ESOP)*.

Asumu Takikawa, Daniel Feltey, Ben Greenman, Max S. New, Jan Vitek, and Matthias Felleisen. 2016. Is Sound Gradual Typing Dead? *SIGPLAN Not.* 51, 1 (Jan. 2016), 456–468.

The Computer Language Benchmarks Game 2021. The Computer Language Benchmarks Game. https://benchmarksgame-team.pages.debian.net/benchmarksgame/index.html.

Sam Tobin-Hochstadt and Matthias Felleisen. 2006. Interlanguage Migration: From Scripts to Programs. In *Dynamic Languages Symposium (DLS)*.

Sam Tobin Hochstadt and Matthias Felleisen. 2008. The Design and Implementation of Typed Scheme. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*.

Panagiotis Vekris, Benjamin Cosman, and Ranjit Jhala. 2015. Trust, but Verify: Two-Phase Typing for Dynamic Languages. In *European Conference on Object-Oriented Programming (ECOOP)*.

Jiayi Wei, Maruth Goyal, Greg Durrett, and Isil Dillig. 2020. LambdaNet: Probabilistic Type Inference using Graph Neural Networks. In *International Conference on Learning Representations (ICLR)*.

Andrew K. Wright and Robert Cartwright. 1997. A Practical Soft Type System for Scheme. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 19, 1 (Jan. 1997), 87–152.