Taming Type Annotations in Gradual Typing

JOHN PETER CAMPORA, University of Louisiana at Lafayette, USA
SHENG CHEN, University of Louisiana at Lafayette, USA

Gradual typing provides a methodology to integrate static and dynamic typing, harmonizing their often conflicting advantages in a single language. When a user wants to enjoy the advantages of static typing, most gradual languages require that they add type annotations. Many nontrivial tasks must be undertaken while adding type annotations, including understanding program behaviors and invariants. Unfortunately, if this is done incorrectly then the added type annotations can be wrong–leading to inconsistencies between the program and the type annotations. Gradual typing implementations detect such inconsistencies at runtime, raise cast errors, and generate messages. However, solely relying on such error messages for understanding and fixing inconsistencies and their resulting cast errors is often insufficient for multiple reasons. One reason is that while such messages cover inconsistencies in one execution path, fixing them often requires reconciling information from multiple paths. Another is that users may add many wrong type annotations that they later find difficult to identify and fix, when considering all added annotations.

Recent studies provide evidence that type annotations added during program migration are often wrong and that many programmers prefer compile-time warnings about wrong annotations. Motivated by these results, we develop *exploratory typing* to help with the static detection, understanding, and fixing of inconsistencies. The key idea of exploratory typing is that it systematically removes dynamic types and explores alternative types for static type annotations that can remedy inconsistencies. To demonstrate the feasibility of exploratory typing, we have implemented it in PyHound, which targets programs written in Reticulated Python, a gradual variant of Python. We have evaluated PyHound on a set of Python programs, and the evaluation results demonstrate that our idea can effectively detect inconsistencies in 98% of the tested programs and fix 93% of inconsistencies, significantly outperforming pytype, a widely used Python tool for enforcing type annotations.

CCS Concepts: • **Theory of computation** → *Program analysis.*

Additional Key Words and Phrases: gradual typing, case errors, variational types

ACM Reference Format:
John Peter Campora and Sheng Chen. 2020. Taming Type Annotations in Gradual Typing. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 191 (November 2020), 30 pages. https://doi.org/10.1145/3428259

1 INTRODUCTION

Static and dynamic typing have different strengths and weaknesses. Static typing enforces strong program correctness guarantees and facilitates program maintenance, but has restricted flexibility and typically delivers no/little runtime feedback for ill-typed programs. Dynamic typing enables fast prototyping and is more flexible, but provides few guarantees about program correctness and the lack of type information hinders program maintenance [Tobin-Hochstadt et al. 2017]. Due to their complementary advantages, integrating both typing disciplines into single languages has been an active research theme, including soft typing [Aiken and Murphy 1991; Cartwright and Fagan 1991], completed dynamic typing [Henglein 1994], partial types [Thatte 1988], dynamic types

Authors' addresses: John Peter Campora, CACS, University of Louisiana at Lafayette, USA, campora@louisiana.edu; Sheng Chen, CACS, University of Louisiana at Lafayette, USA, chen@louisiana.edu.

2475-1421/2020/11-ART191
https://doi.org/10.1145/3428259

and type cases [Abadi et al. 1991], hybrid type checking [Knowles and Flanagan 2010], gradual typing [Siek and Taha 2006], and migratory typing [Tobin-Hochstadt and Felleisen 2006].

The last decade has seen a rapid development of gradual typing and migratory typing. In academia, researchers have studied the interaction of gradual typing with various language features[Ahmed et al. 2011, 2017; Bañados Schwerter et al. 2014, 2016; Bader et al. 2018; Castagna and Lanvin 2017; Disney and Flanagan 2011; Fennell and Thiemann 2013; Garcia et al. 2014; Herman et al. 2010; Igarashi et al. 2017b,a; Ina and Igarashi 2011; Jafery and Dunfield 2017; Lehmann and Tanter 2017; Sergey and Clarke 2012; Siek and Taha 2007; Siek et al. 2015b; Toro et al. 2019; Wolff et al. 2011; Xie et al. 2018]. In industry, several languages supporting gradual or optional typing have been created. For example, static type checking has been added to Dart, JavaScript (TypeScript), PHP (Hack), and Racket (Typed Racket), and dynamic types have been added to C#.

1.1 Wrong Type Annotations and Their Consequences

A main goal of gradual typing is to allow users to easily *migrate* programs–that is adding or removing type annotations to enjoy more advantages of static or dynamic typing, respectively. Removing type annotations is an easy task, particularly as the gradual guarantee [Siek et al. 2015a] specifies that gradual typing should preserve program behaviors as type annotations are removed. Adding type annotations, however, is a much more challenging task that entails many nontrivial activities, including reading and understanding programs, abstracting program states, and rediscovering program invariants [Tobin-Hochstadt and Felleisen 2006; Tobin-Hochstadt et al. 2017]. As a result, it is quite common that the added type annotations may contain mistakes (the end of this subsection discusses previous research supporting this claim), which lead to inconsistencies between type annotations and the program. Due to the permissiveness of gradual typing, such inconsistencies may not be caught at compile time and will instead be detected at runtime [Siek and Taha 2006].

```
1  def idD(x): return x
2
3  def equal(fst : String) -> (String -> Bool):
4      def impl(snd : String) : return fst == snd
5      return impl
6
7  def asciify(val : String, mkStr : String -> String,
8              opts : ★, callback : String -> ★):
9      return callback(mkStr(val) + opts)
10
11 asciify(idD('B'), idD, '3-d',equal('B'))
12 # str has the type ★ -> String
13 asciify(idD(138), str, 'pyramid', equal('138'))
14 asciify(idD(False), str, 'pyramid', equal('False'))
```

Fig. 1. A Reticulated Python program mixing static and dynamic typing. Expressions relying on parameters and variables without type annotations (or annotated with ★) are dynamically typed, and other expressions are statically typed.

There are two kinds of type inconsistencies in gradual programs: static and dynamic inconsistencies that can and cannot be caught by a gradual type system statically, respectively. This paper focuses on the latter kind. As a result, inconsistencies in the paper can always be understood as runtime inconsistencies or cast errors.

As an example of an inconsistency, consider the program in Figure 1, adapted from [Williams et al. 2017] and asciify [2018]. Without type annotations, this program runs correctly. With type annotations, the program is still statically well typed because `idD(138)` and `idD(False)` have unknown types (also known as *dynamic types*, written as ★), which can be validly passed to any functions in gradual type systems. However, the annotations cause several cast errors. For example, `idD(138)` and `idD(False)` produce values whose runtime types are `Int` and `Bool`, respectively. When they are passed into `asciify` in the last two lines of Figure 1, two cast errors will be raised because

these values do not have the type `String`, `asciify`'s first parameter type (technically only the one involving `138` is raised, but if the execution continued the other would also be raised).

Gradual language implementations [Siek et al. 2015a] catch such inconsistencies (violations of type invariants) dynamically. Moreover, they use blame tracking [Ahmed et al. 2011; Siek et al. 2015a; Wadler and Findler 2009] to pinpoint program expressions that caused the violations. For example, when running the program in Figure 1, Reticulated Python [Vitousek et al. 2017] reports the following message, indicating that the dynamically typed value, `138`, caused the cast error.

```
15  retic.guarded.CastError:
16  asciify.py:15:4: Expected argument of type String but value 138 was provided instead.
```

However, an approach that can statically detect such inconsistencies and provide fix suggestions is beneficial for the following reasons. First, as cast errors are detected at runtime, they may be undetected by programmers if they are not covered during developing and testing. They may be encountered after the program is deployed and observed by users who have no expertise in fixing such errors. Static approaches mitigate this issue. Second, handling cast errors at compile time can assist program migration towards utilizing more static typing, when compared to detecting such errors at runtime. Without the compile time detection of errors, programmers may introduce many incorrect type annotations. They then must rerun the program multiple times to discover each cast error, which could be avoided by immediately warning users about wrong type annotations.

Third, while blame tracking covers information for an inconsistency from a particular execution path, fixing the inconsistency by changing wrong type annotation may require reconciling many potential execution paths, which a static approach could provide quite easily. Consider, for instance, the above message from Reticulated. This message mentions only the function call on line 13, and following this message one may change the type of `val` from `String` to `Int` to fix the cast error. Unfortunately, this fix will lead to another cast error in the program, namely on line 11 because `idD('B')` is not a value of `Int`, the new type annotation for `val`, at runtime. In fact, to correctly fix the cast error, one needs to reconcile the calls on lines 11 through 14, which becomes a more significant undertaking, especially when relevant calls are scattered throughout larger programs.

Empirical facts and user preferences Several studies have shown that type annotations are error prone. Wu and Chen [2017] performed a study on a program database that contains more than 55,000 Haskell programs, which logged student interactions of learning functional programming. The results showed that among 2,757 programs that have type errors, 30% were caused by mistakes in type annotations. In Typed Racket, St-Amour and Toronto [2013] indicated that wrong types provided by the language developers were a main source (around 17.7%) for the 576 reported bugs. In testing whether the type definition files in TypeScript are consistent with the (underlying Javascript) libraries and test cases, Williams et al. [2017] observed that among 122 checked libraries 62 libraries contain inconsistencies. Given that definition files are added later, it could be understood that 50.8% of type definition files contain errors. In a separate study of testing the inconsistencies between Typescript declaration files and libraries, Kristensen and Møller [2017b] observed that numerous inconsistencies were detected in 59 libraries, and they further noted that "some declaration files contain dozens of actual errors".

Overall, these facts indicate that migrating gradual programs towards more static could easily introduce mistakes in type annotations. This coincides with the perspective shared by Siek et al. [2015a], who noted that static or dynamic tools are expected to support migrating programs. In above, we have given a few reasons why static support is more preferable than the dynamic one. The insights from a recent survey by Tunnell Wilson et al. [2018] confirm our view, where programmers indicated that they prefer more compile-time error reporting for wrong type annotations. Our goal

in this paper is to provide a static approach that detects and explains cast errors and recommend fixes to them when they are caused by wrong type annotations.

1.2 Challenges in Detecting and Fixing Wrong Type Annotations

To detect inconsistencies between type annotations and the program, it is insufficient to solely reason about the compile-time types assigned to expressions because dynamic types (⋆s) suppress useful type information. For example, in gradual typing, the ascription `1:⋆` has the type ⋆, which suppresses the "real" type `Int` for `1` at compile time. In the `asciify` example, the function `idD` has the type $\star \rightarrow \star$, making `idD(138)` and `idD(False)` have types ⋆. As values having the type ⋆ can be passed to any function, no errors are detected when passing `idD(138)` and `idD(False)` to `asciify`, though its annotations expects values to have the type `String`.

Therefore, to uncover inconsistencies at compile time, we must find an approach to recover information suppressed by ⋆s. We can achieve this by using more precise types, rather than ⋆s, for expressions. For example, since the return type of `idD` is the same as its parameter type, we can compute that `idD(138)` and `idD(False)` have the types `Int` and `Bool`, respectively. These two types now enable us to detect type conflicts with `String`, the type of the first parameter of `asciify`. In general, we can detect inconsistencies by recovering as much type information from dynamically typed expressions as possible.

After detecting inconsistencies at compile time, a real challenge is developing a method for automatically computing fixes to remove them. Inconsistencies can be fixed by removing or changing type annotations for parameters. For example, the inconsistencies caused by `idD(138)` and `idD(False)` can be removed by changing `val`'s `String` type annotation to ⋆. However, determining the exact parameters whose types have to be changed and how to change them is nontrivial. The simple strategy that removes all type annotations will eliminate inconsistencies but is highly undesirable since it counters the goal of migrating programs toward more static.

Another plausible strategy is to remove each type annotation individually and check whether the removal will eliminate the inconsistency. Unfortunately, this strategy also does not work because inconsistencies remain unless all relevant static type annotations are removed. For example, only when the type annotations for both `val` and `mkStr` are simultaneously removed can `asciify` be free of inconsistencies. In fact, there appears to be no good strategy for finding the smallest set of parameters for which type annotations need to be removed to fix the inconsistency. The only viable strategy is enumerating all the combinations of static parameters and removing their type annotations until the inconsistency no longer exists, leading to a problem of exponential complexity.

1.3 A Solution Based on Variational Typing

To address the exponential complexity problem above, our key observation is that the programs yielded from two different ways of changing[1] static types differ only slightly, accounting for a small portion of the whole program. For example, the programs produced by changing `fst`'s type annotation versus that for changing `snd`'s annotation are almost the same. In particular, the typing processes for these two programs differ only in typing the expression `fst == snd` and the three calls to `equal`. The typing for other parts, including the functions `idD` and `asciify`, is the same.

Motivated by this observation, we propose employing *variational typing* [Chen et al. 2014a] to reuse computations when type checking similar programs produced by changing different combinations of static type annotations. A variational type succinctly encodes a large set of related types. Thus, while conventionally an expression can be assigned only one type, each expression may be assigned multiple types in variational typing [Chen et al. 2014a]. Variational typing allows

[1]We compute the static type to change to through type inference, which will be discussed in Section 4.5.

us to check a program part with multiple types while visiting it just once, thereby reducing the exponential complexity of typing all possible programs.[2]

Variational typing helps compute fixes to wrong type annotations as follows. For each static parameter, we have two possibilities: the type annotation is either not wrong or wrong. Correspondingly, we either do not need to change the static type annotation on the parameter under consideration, or we need to replace it with another static type[3]. We use a unique variation to encode these two possibilities for each static parameter, with the first alternative of the variation being the type annotation of the parameter and the second being another static type. After the program is typed, we can find the variations whose second alternatives must be used to remove inconsistencies in the program. Since each variation can be uniquely traced back to the parameter, we can find the smallest set of static parameters to fix the inconsistency. We give more details about this process through an example in Section 3.

To illustrate the difference between naively searching the exponential space of annotation change versus using variational typing, consider again the `asciify` example. Since it has 8 static types (7 `Strings` plus 1 `Bool`), we need to type-check and compare $2^8 = 256$ programs. In contrast, with variational typing, we type check just one program with 8 variations, which variational typing can handle with ease, as it can type programs with tens of thousands of variations efficiently [Chen et al. 2012b, 2014b].

Overall, the fundamental idea of the solution is to *explore* the types that can be assigned to dynamic parameters when ⋆s are uncovered and alternative types that can be assigned to static parameter to remove inconsistencies. For this reason, we name our solution *exploratory typing*. To test the viability of exploratory typing, we have developed a tool named PyHound in Reticulated Python [Vitousek et al. 2014, 2017] for repairing programs with wrong type annotations at compile time in Python. For the `asciify` example, PyHound generates the following message.

```
17  The program has type inconsistencies at the following expressions:
18      asciify(idD(138), str, 'pyramid', equal('138'))
19      asciify(idD(False), str, 'pyramid', equal('False'))
20    because:
21      Int, the type of idD(138) at runtime, conflicts with
22          String, the type annotation for val (the first parameter of asciify)
23      Bool, the type of idD(False) at runtime, conflicts with
24          String, the type annotation for val (the first parameter of asciify)
25  The inconsistencies are cloaked by:
26      The ⋆ for the type annotation of idD
27  Possible fixes:
28      (1) Change the expressions mentioned on lines 18 through 19, or
29      (2) Change the type annotations for asciify as follows
30          Replace String for val with ⋆ and
31          Replace String -> String for mkStr with ⋆ -> String
```

This message consists of three parts, for statically detecting, understanding, and fixing runtime inconsistencies, which we refer to as S1, S2, and S3, respectively.

S1 This part of the message, including lines 17 through 24, gives the type annotations and the expressions that lead to runtime inconsistencies. The user can exploit this information to fix the inconsistencies if he/she believes the errors are in expressions. In this particular example, `idD(138)`

[2]Other type constructs, such as union types, could also assign multiple type variants to a single expression. However, unlike variational types, their goals are not to reuse computations. For a detailed discussion, please see Chen et al. [2014a].
[3]Our messages sometimes suggest to change the static type to a ⋆, which happens when the suggested type cannot be represented in the gradual language.

and `idD(False)` have to be changed to have the type `String`. This can be achieved, for example, by changing `138` and `False` to `'138'` and `'False'`, respectively.

S2 The second part of the message, including lines 25 and 26, conveys that the inconsistencies are *cloaked* by the type annotation for `idD`. We say ⋆s are *cloaking* if their presence prevent the inconsistencies from being detected by gradual type systems at compile time. This happens because ⋆s suppress real types of their underlying values and allow such values to slip into contexts expecting values of other types. Note that not all ⋆s are cloaking, and we will see such an example in Section 3.

S3 The third part of the message, including lines 27 through 31, presents a recommendation for fixing the inconsistency. Since the inconsistency could be caused by the two function calls to `asciify` or the type annotation for `asciify`, our fix suggestion lists these two possibilities. Specifically, line 28 specifies expressions to be fixed if the programmer believes the inconsistency was caused by expressions. Lines 29 through 31 specify changes needed to type annotations if the user believes that the inconsistencies are caused by the type annotations. The recommendation suggests to use ⋆ for `String` solely because Reticulated Python does not support polymorphic types. Should Reticulated support that, our suggestion would be change `String` to α and change `String -> String` to α `-> String`, where α is a type variable. In general, PyHound finds the most precise static type to fix cast errors and uses ⋆ when that type cannot be represented by the gradual language. We give more details about finding fixes in Section 5.2.

We observe that the message for changing the type annotations is much more concrete than that for changing the expressions. The reason is that there are too many possibilities in changing expressions to fix the type inconsistency. Instead, the possibilities for type changes are fewer and can often be inferred by making good use of the type information of relevant expressions. Nevertheless, reconciling type information from different execution paths could be a challenging task (Section 1.1), and one main goal of this paper is to automate this task.

1.4 Relation with Previous Work and Contributions of This Work

The earliest work that tries to identify errors in type annotations dates back to Braßel [2004]. However, that work dealt with static typing with type inference but not gradual typing, and so did not face challenges from this paper. It also did not try to compute fixes. There have been some static and dynamic analyses to detect mistakes in Typescript definition files [Feldthaus and Møller 2014; Kristensen and Møller 2017b; Williams et al. 2017]. This paper goes further to explain what dynamic types have cloaked type inconsistencies from being detected by gradual type systems and to compute recommendations for fixing wrong type annotations. Finally, Campora et al. [2018b] developed a solution for adding type annotations to gradual programs written in Hindley-Milner extended with gradual types [Garcia and Cimini 2015]. However, it does not aim to identify cast errors in the programs, nor does it compute fixes for them. For a program with cast errors, it may suggest adding type annotations that introduce further mistakes into the program. For example, for the expression e3 in Section 3, the work in Campora et al. [2018b] suggests adding a `Bool` annotation to the parameter x, making that expression even more incorrect. In contrast, this work can detect the problem in the expression and find a fix to it. Moreover, this paper develops mechanisms for handling tricky language features that are absent in Campora et al. [2018b], such as conditionals having different branch types.

In developing an approach for statically detecting, understanding, and fixing runtime inconsistencies, this paper makes the following contributions.

(1) In Section 1, we motivated the importance of detecting, understanding, and fixing inconsistencies at compile-time. We also outlined an approach to provide such supports.

(2) In Section 4, we develop *exploratory typing* for analyzing the dynamic type safety of gradual programs. The type system handles some ubiquitous yet challenging features used in dynamically-typed languages, including conditional statements with different branch types and expressions whose types cannot be determined at compile-time. We prove that our type system is correct in Section 4.4.
(3) In Section 5, we provide methods to detect, understand, and fix inconsistencies after exploratory typing finishes. We show the correctness of them.
(4) In Section 7, we evaluate PyHound using benchmarks from the literature [Campora et al. 2018a; Vitousek et al. 2017]. Our evaluation on 282 programs with inconsistencies demonstrates that PyHound can effectively support its intended goal and can scale to large programs.

We present background in Section 2, illustrate the idea of exploratory typing through an informal example in Section 3, discuss related work in Section 8, and conclude in Section 9.

2 BACKGROUND

In this section, we review gradual typing and variational typing, which form the cornerstones for the technical developments in subsequent sections. For simplicity we will now use simple examples written in the gradually typed lambda calculus developed by Siek and Taha [2006].

2.1 Gradual Typing

As shown earlier, gradual typing allows the interoperation between dynamically and statically typed code. At the heart of this interoperation is a *consistency* relation (denoted by $\sim$). The relation $G_1 \sim G_2$ states that two types G_1 and G_2 are consistent if no parts of them disagree statically. The dynamic type $\star$ is trivially consistent with any type because it has no static parts. Primitive static types (such as `Int`) are consistent with themselves. Two function types are consistent when their domains and codomains are consistent. Therefore, we have $\texttt{Int} \sim \texttt{Int}$, $\star \sim \texttt{Int}$, and $\star \to \texttt{Int} \sim \texttt{Int} \to \texttt{Int}$. However, $\star \to \texttt{Bool}$ and $\texttt{Int} \to \texttt{Int}$ are not consistent since their codomains are inconsistent.

The equality checking in static typing is weakened to the consistency checking in gradual typing [Garcia et al. 2016]. Casts will be inserted into the places where two types are consistent, but not equal, to ensure that types are indeed equal at runtime. For example, consider the expression $(\lambda y : \texttt{Bool}.y)\ ((\lambda z : \star.z)\ 42)$. In this expression, a cast is inserted at the innermost application to check whether the argument `42` has the type $\star$ or not at runtime. At the outermost application, a cast is inserted to check if the argument has the type `Bool`, the parameter type of the function $\lambda y : \texttt{Bool}.y$. If a check fails, then a cast error will be raised. Thus, this program raises a cast error at the outermost application.

2.2 Variational Typing

Variational typing [Chen et al. 2012b] is used to type check variational programs. The goal of *variations* is to succinctly represent a large set of different but closely related expressions or types. For example, the following expression represents two expressions `succ 0` and `succ (1 : ⋆)`.

$$\texttt{succ}\ A\langle \texttt{0}, \texttt{1} : \star \rangle \tag{e1}$$

This expression contains a choice named A, with the *first alternative* `0` and the *second alternative* $\texttt{1} : \star$. We use the meta-variable d to range over choice names. A variational program may contain any number of choices, and their names do not have to be unique. A program is *plain* if it contains no choices. A *selector*, written as $d.i$, consists of a choice name d and an alternative index i. Variations can be eliminated through the *selection* process, denoted as $\lfloor o \rfloor_{d.i}$, which replaces all choices named d in the variational object (expression or type) o with their ith alternative. For example we have:

$$\lfloor \texttt{Int} \rfloor_{A.1} = \texttt{Int} \qquad \lfloor A\langle \texttt{0}, 1 : \star \rangle \rfloor_{A.2} = 1 : \star$$
$$\lfloor A\langle \star, \texttt{Int} \rangle \to A\langle \texttt{Int}, \star \rangle \rfloor_{A.1} = \star \to \texttt{Int} \qquad \lfloor A\langle \star, \texttt{Int} \rangle \to B\langle \texttt{Int}, \star \rangle \rfloor_{A.2} = \texttt{Int} \to B\langle \texttt{Int}, \star \rangle$$

The examples in the second row indicate that variations with the same name are synchronized (both As in the left column are eliminated) and those with different names are independent (only A, but not B, in the right column is eliminated). A *decision*, written as δ, is a set of selectors. Selection extends naturally to decisions by recursively applying the selectors in the decision.

To type the expression e1, we note that the variation is directly lifted to the type level so that the argument has the type $A\langle \texttt{Int}, \star \rangle$. The succ function has the type $\texttt{Int} \to \texttt{Int}$. To type the expression e1, we need to match the parameter type Int with the type of the argument $A\langle \texttt{Int}, \star \rangle$. Since these types contain both a $\star$ and a variation A, the traditional typing rule for applications does not directly apply, and neither do the typing rules for gradual typing [Siek and Taha 2006] or variational typing [Chen et al. 2014b]. Campora et al. [2018b] introduced a type *compatibility* relation, written as $\approx$, to address this issue. The $\approx$ relation combines the consistency relation in gradual typing and the type equivalence relation in variational typing. Intuitively, two types are compatible if selecting them with any decision that eliminates all variations yields two consistent types. Following this intuition, the type of the argument ($A\langle \texttt{Int}, \star \rangle$) and the parameter type (Int) are compatible. As a result, the expression e1 is well typed and has the type Int.

2.3 Error Tolerance

Next let us change 0 to False in the expression e1 and retype it.

$$\texttt{succ}\ A\langle \texttt{False}, 1 : \star \rangle \tag{e2}$$

First, the argument now has the type $A\langle \texttt{Bool}, \star \rangle$, which is incompatible with Int, the parameter type of succ. Traditionally, this would cause the typing process to terminate with an error. This is, however, undesirable in variational typing because we are interested in both knowing typing results for well typed variants and determining which variants have type errors. Under early termination, we are not able to generate explanations for understanding inconsistencies nor are we able to compute fixes for them. Chen et al. [2012b] avoided the early termination problem with the idea of error-tolerant typing and Campora et al. [2018b] adapted that solution to work with gradual types. The main idea of their solution is extending the typing process $\Gamma \vdash e : V$ with a pattern π to indicate the validity of the process, yielding $\pi; \Gamma \vdash e : V$. Here Γ is a type environment and V denotes a variational type (see Section 4.1).

A pattern can be $\bot$, indicating that the typing process is invalid, $\top$, indicating the process is valid, or a variation between two patterns that recursively indicate the validity in typing the variational program. Recall that in Section 1.3 we said that, for any expression, our typing result encodes all possibilities for understanding and fixing type inconsistencies (by changing static type annotations). Intuitively, within those possibilities, some are well-typed, meaning that changing the type annotation in the corresponding ways could indeed fix the inconsistencies, and some are ill-typed, meaning that the corresponding changes could not fix the inconsistencies. We can extract the well-typed possibilities by collecting all possibilities whose typing patterns are $\top$ and ignoring the possibilities whose patterns are $\bot$. Our implementation, which we present in Section 5, takes a more efficient approach, by exploring the relations between patterns.

With the new typing relation, we can type the expression e2 as follows. First, we have $\bot; \Gamma \vdash \texttt{False} : \texttt{Int}$. Note that the pattern is $\bot$, since the process of assigning the type Int to False is invalid. Similar, we have $\top; \Gamma \vdash (1 : \star) : \star$ since assigning $\star$ to $1 : \star$ is valid. Combining the previous two typing judgments (by embedding the patterns, the expressions, and the types into the variation $A\langle , \rangle$), we have $A\langle \bot, \top \rangle; \Gamma \vdash A\langle \texttt{False}, 1 : \star \rangle : A\langle \texttt{Int}, \star \rangle$. Since $A\langle \texttt{Int}, \star \rangle$, the type of the argument in the expression e2, is compatible with Int, the parameter type of succ, we have the

following typing result for the expression e2, $A\langle\bot, \top\rangle; \Gamma \vdash$ e2 : `Int`. Based on the typing pattern for e2, only the result in $A.2$ is valid, and that in $A.1$ is invalid. This is desirable since the expression at $A.2$ is `succ (1 :` $\star$`)` and that at $A.1$ is `succ False`.

Like with the typing relation, we can similarly decorate other relations with typing patterns for error tolerance. To illustrate, consider decorating the type compatibility relation $\approx$ with a pattern to form a new relation $\approx_\pi$, which could be applied to incompatible types. For example, the types `Int` and $B\langle\star, \texttt{Bool}\rangle$ are incompatible because `Int` is not compatible with the second alternative of the B variation. However, the two types can be considered as compatible if we restrict the relation with the pattern $B\langle\top, \bot\rangle$ since the pattern already indicates that the validity of compatibility is only required for the first alternative of B (because the pattern there is $\top$) but not the second alternative (because the pattern there is $\bot$). We express this result as `Int` $\approx_{B\langle\top,\,\bot\rangle} B\langle\star, \texttt{Bool}\rangle$.

3 FIXES THROUGH AN INFORMAL EXAMPLE

In this Section, we give a detailed description of how we use exploratory typing to facilitate S1 through S3 for detecting, understanding, and fixing dynamic inconsistencies, by typing and analyzing the program below. The program locations (ℓ_1, ℓ_2, and ℓ_3) are needed for the formal presentation of the type system (in Section 4) and can be ignored for now.

$$(\lambda^{\ell_1} y\colon \texttt{Bool}.y)\ ((\lambda^{\ell_2} x\colon \star.x)\ ((\lambda^{\ell_3} z\colon \star.z)\ 42)) \tag{e3}$$

As mentioned in Section 1.3, we detect inconsistencies by recovering type information hidden by uses of $\star$ at compile-time. We fulfill that purpose by creating a variation for each dynamic parameter with the first alternative being a $\star$ and the second alternative being an appropriately recovered type for the parameter. We call such variations *dynamic variations* and use the metavariable D to range over them. Following this idea, we assign the dynamic variation $D_1\langle\star, \texttt{Int}\rangle$ to the parameter x and $D_2\langle\star, \texttt{Int}\rangle$ to z. The type `Int` in the second alternative can be computed through type inference, which we discuss in Section 4.5. Note that D_1 and D_2 represent two distinct variations, allowing us to directly connect typing results to parameters, for use in determining the set of cloaking $\star$s mentioned in S2.

For static parameters, we similarly create variations in the *static domain* S. For example, for the parameter y we assign a static variation whose first alternative is `Bool`, the annotation given to y in the program. What should the second alternative be? One may suggest to use a $\star$. However, that is undesirable for several reasons. First, using a $\star$ means removing that static type annotation, which counters the goal of migrating gradual programs toward utilizing more static checking. Second, using a $\star$ may falsely compute a fix where none is possible.

For example, consider `(`λx`: Int.`x `+ length (`x `:` $\star$`))` 3. This expression is statically well typed because the ascription $x : \star$ assigns the type $\star$ to x, making the type of $x : \star$ consistent with the parameter type of `length`. Removing the ascription, we can detect that this program contains a runtime inconsistency since the program is ill typed. We can provide what appears a fix by changing the `Int` annotation for x to $\star$. This will make the program statically well typed but an inconsistency remains at the call to `length` (which expects lists) and thus this is not a proper fix. In fact, there is no fix possible for this program by solely changing annotations.

In fact, we should use a static type for the second alternative because a static type ensures that all of its uses are consistent. Following this idea, we use `Int` (Again computed through type inference) for y and so the variational type for it should be $S_1\langle\texttt{Bool}, \texttt{Int}\rangle$. Note that variation names S_i are drawn from the static domain S, and D_i are drawn from the dynamic domain D. The use of two different domains is necessary since each supports different aspects of the repair process for inconsistencies.

We list the typing process for the expression e3 in Figure 2, whose resulting pattern π_{e3} and type E_{e3} are as follows.

$$\pi_{e3} = S_1\langle D_1\langle \top, \bot\rangle, \top\rangle \qquad E_{e3} = S_1\langle \texttt{Bool}, \texttt{Int}\rangle$$

The pattern indicates that a typing conflict is detected at $\{S_1.1, D_1.2\}$ for this expression.

From π_{e3} and E_{e3}, we can compute repair steps (S1 through S3). As discussed in Section 2.2, we can eliminate variations from variational types and patterns by selecting them with decisions. Supporting each repair step corresponds to finding decisions that eliminate typing patterns in a specific way, and we discuss the process for each repair step below.

Expressions	Types
$\lambda^{\ell_3} z\colon \star.z$	$D_2\langle\star, \texttt{Int}\rangle \to D_2\langle\star, \texttt{Int}\rangle$
42	$\texttt{Int}$
$(\lambda^{\ell_3} z\colon \star.z)\ 42$	$D_2\langle\star, \texttt{Int}\rangle$
$\lambda^{\ell_2} x\colon \star.x$	$D_1\langle\star, \texttt{Int}\rangle \to D_1\langle\star, \texttt{Int}\rangle$
$(\lambda^{\ell_2} x\colon \star.x)\ ((\lambda^{\ell_3} z\colon \star.z)\ 42)$	$D_1\langle\star, \texttt{Int}\rangle$
$\lambda^{\ell_1} y\colon \texttt{Bool}.y$	$S_1\langle\texttt{Bool}, \texttt{Int}\rangle \to S_1\langle\texttt{Bool}, \texttt{Int}\rangle$
Expression e3	$S_1\langle\texttt{Bool}, \texttt{Int}\rangle$

Fig. 2. Variational types for typing different subexpressions of e3. The patterns for the first six rows are $\top$ and that for the last row is $S_1\langle D_1\langle \top, \bot\rangle, \top\rangle$.

S1 To decide whether the program contains a runtime inconsistency, we start by selecting the pattern with the decision that contains $D_i.2$ for each D_i created. This replaces the $\star$s that suppress types with proper static types for use in detecting errors statically. We also select with $S_i.1$ for each S_i because we are detecting whether runtime inconsistencies exist under the existing type annotations. Following this idea, the decision for detecting the error in the expression e3 is $\delta = \{D_1.2, S_1.1\}$. Since selecting the pattern π_{e3} with δ yields a $\bot$, we conclude that e3 contains runtime inconsistencies.

S2 To help compute cloaking $\star$s, we first observe that: (1) For each static variation S_i we should select the pattern (π_{e3}) with $S_i.1$ to keep the original static type information. (2) For each dynamic variation D_i, if the corresponding $\star$ helps hide an inconsistency then the pattern is $\top$ when selecting $D_i.1$ and is $\bot$ in $D_i.2$. In other words, $D_i\langle\top, \bot\rangle$ should be in the result pattern. Recall from Section 2.3 that a $\top$ and a $\bot$ denote the typing is valid and invalid, respectively. Returning to the expression e3, with observation (1), we can reduce π_{e3} to $D_1\langle\top, \bot\rangle$. With observation (2), we derive that the $\star$ for x, the parameter that corresponds to D_1, is cloaking.

S3 To fix inconsistencies, we need to produce the decision that (1) contains $D_i.2$ for all D_is and $S_i.2$ for any number of S_is and (2) yields $\top$ when selecting the pattern with it. Essentially, this communicates whether the newly found type information for statically typed parameters ($S_i.2$) agrees with that for the recovered dynamic ($D_i.2$) type information. In our example, there is only one decision, $\delta = \{D_1.2, S_1.2\}$, that satisfies (1) and (2). Consequently, we conclude that changing the type annotation of y, the parameter that corresponds to S_1, to `Int` can fix the inconsistencies.

4 EXPLORATORY TYPING

The informal example in Section 3 illustrates that the fundamental idea of exploratory typing is *exploring* the typing results for a large set of programs that can be generated by replacing dynamic and static parameters with alternative types. We present the syntax of exploratory typing in Section 4.1, introduce all of the different variational domains in Section 4.2, present the typing rules in Section 4.3, and investigate its properties in Section 4.4.

4.1 Syntax

We present the syntax of expressions, types, and various typing judgment elements in Figure 3. The source language is a straightforward extension of the gradually typed lambda calculus [Siek and

Term variables x, y, z Value constants c
Type variables α Type constants γ Program locations ℓ

$$
\begin{array}{lrcl}
\text{Expressions} & e & ::= & c \mid x \mid \lambda^{\ell} x\colon G.e \mid e\ e \mid e_o^{\ell} \mid \textbf{let}\ x = e\ \textbf{in}\ e \mid \textbf{if}^{\ell}\ e\ \textbf{then}\ e\ \textbf{else}\ e \\
\text{Static types} & T & ::= & \gamma \mid \alpha \mid T \to T \mid F\langle T, T\rangle \\
\text{Gradual types} & G & ::= & \gamma \mid G \to G \mid \star \mid F\langle G, G\rangle \\
\text{Variation domains} & d & ::= & F \mid D \mid S \mid O \\
\text{Variational types} & V & ::= & \gamma \mid \alpha \mid V \to V \mid d\langle V, V\rangle \\
\text{Exploratory types} & E & ::= & \gamma \mid \alpha \mid E \to E \mid \star \mid d\langle E, E\rangle \\
\text{Type Schemes} & \sigma & ::= & E \mid \forall \overline{F}.E \\
\text{Type environment} & \Gamma & ::= & \varnothing \mid \Gamma, x \mapsto E \\
\text{Exploratory map} & \Omega & ::= & \varnothing \mid \Omega, \ell \mapsto E \\
\text{Program update} & K & ::= & \varnothing \mid K, \ell \mapsto G \\
\text{Typing patterns} & \pi & ::= & \bot \mid \dagger \mid \top \mid d\langle \pi, \pi\rangle
\end{array}
$$

Fig. 3. Syntax of exploratory typing.

Taha 2006], with constants (c), conditionals, and opaque expressions (e_o). We introduce program locations (ℓ) to record where variations are introduced. The opaque expression can be instantiated with language features that make type-based exploratory analysis difficult, like the `eval` expression. We discuss typing opaque expressions in Section 4.2.

The syntax of types is stratified into several kinds. In addition to conventional constructs, static types (T) contain F variations. The F domain is used to more precisely type conditionals, and we discuss it in detail in Section 4.2. While static types and gradual types (G) do not contain choices other than F, variational (V) and exploratory types (E) do. We use the meta-variable d to range over variations from the D, S, F, and O domains. We have seen the first two domains in Section 3. The O domain is used to safely associate types to opaque expressions. Type schemes quantify over F variations, and we will elaborate on their purpose when discussing the typing of let-expressions in Section 4.3.

The definition of type environments Γ is standard. The exploratory map (Ω) records where and what variations are introduced during the typing process. For example, let Ω_{e3} be the exploratory map in typing the expression e3 in Section 3, then

$$\Omega_{e3} = \{\ell_2 \mapsto D_1\langle\star, \texttt{Int}\rangle, \ell_1 \mapsto S_1\langle\texttt{Bool}, \texttt{Int}\rangle, \ell_3 \mapsto D_2\langle\star, \texttt{Int}\rangle\}.$$

The main goal of introducing K is to establish the correctness of our type system, and we leave the discussion of it to Section 4.4. The syntax for typing patterns (π) extends those in Section 2.2 with a $\dagger$ for typing opaque expressions. We discuss this in Section 4.2.

4.2 Typing Rules Introducing Variational Domains

In this subsection we motivate the needs for four different variational domains and present the typing rules for introducing them in Figure 4. Our typing judgments have the form $\pi; \Gamma \vdash e : E \mid \Omega$, which can be read as: under Γ, the expression e has the type E, with alternative type assignments to parameters recorded in the exploratory map Ω and judgment validity specified in the typing pattern π.

The rule Abs types abstractions. Given a gradual type G for the parameter, we first use the following function *varIntro* to make G variational (the first premise, denoted by E_1) and type the body of the abstraction under the assumption that the parameter has the type E_1 (the second

$$\textsc{Abs}\ \frac{E_1 = \mathit{varIntro}(G) \qquad \pi; \Gamma, x \mapsto E_1 \vdash e : E \mid \Omega}{\pi; \Gamma \vdash \lambda^{\ell} x\colon G.e : E_1 \to E \mid \Omega \cup \{\ell \mapsto E_1\}}$$

$$\textsc{If}\ \frac{\begin{array}{c} F_i \text{ fresh} \qquad F_i\langle \pi_1, \pi_2\rangle; F_i\langle \Gamma_1, \Gamma_2\rangle \vdash e_1 : E_1 \mid \Omega_1 \qquad E_1 \approx_{F_i\langle \pi_1, \pi_2\rangle} \texttt{Bool} \\ \pi_1; \Gamma_1 \vdash e_2 : E_2 \mid \Omega_2 \qquad \pi_2; \Gamma_2 \vdash e_3 : E_3 \mid \Omega_3 \qquad \Omega = \Omega_1 \cup \Omega_2 \cup \Omega_3 \cup \{\ell \mapsto F_i\langle E_2, E_3\rangle\} \end{array}}{F_i\langle \pi_1, \pi_2\rangle; F_i\langle \Gamma_1, \Gamma_2\rangle \vdash \textbf{if}^{\ell}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3 : F_i\langle E_2, E_3\rangle \mid \Omega}$$

$$\textsc{Opaque}\ \frac{O_i \text{ fresh}}{O_i\langle \top, + \rangle; \Gamma \vdash e_o^{\ell} : O_i\langle \star, V\rangle \mid \{\ell \mapsto O_i\langle \star, V\rangle\}}$$

Fig. 4. Typing Rules for introducing variations.

premise). In the conclusion, we record the change $(\ell \mapsto E_1)$ in Ω.

$$\begin{array}{rll} \mathit{varIntro}(T) = & S_i\langle T, V\rangle & \text{where } S_i \text{ fresh} \\ \mathit{varIntro}(\star) = & D_i\langle \star, V\rangle & \text{where } D_i \text{ fresh} \\ \mathit{varIntro}(G_1 \to G_2) = & \mathit{varIntro}(G_1) \to \mathit{varIntro}(G_2) & \end{array}$$

The intuitions on why and how we create variations were given in Section 3. Essentially, for each static type T, we create a static variation S_i (the first case of *varIntro*). For each $\star$, we create a dynamic variation D_i (the second case of *varIntro*). The variation names should be unique (expressed as conditions in *varIntro*) such that we could explore type changes to different parameters independently (Section 2.2). Note that the second alternative in the first two cases of *varIntro* use V, indicating that we may change a static type or a $\star$ to any variational type during exploration (In type inference, the second alternatives will be fresh type variables and will be made concrete based on type constraints from the program (Section 4.5)). As an example of *varIntro*, $\mathit{varIntro}(\star \to \texttt{Bool}) = D_1\langle \star, E_1\rangle \to S_1\langle \texttt{Bool}, E_2\rangle$, where E_1 and E_2 denote any variational gradual type.

The goal of the domain F is to more precisely represent and reason about conditional statements with different branch types. For example, consider the following expression:

$$\texttt{succ\_or\_not} = \lambda^{\ell_1} x\colon \star.\lambda^{\ell_2} y\colon \star.\textbf{if}^{\ell_3}\ x\ \textbf{then}\ \texttt{succ}\ y\ \textbf{else}\ \texttt{not}\ y$$

What type should we assign to the parameter y? We cannot assign a traditional static type since the then branch requires its type to be `Int` whereas the else branch requires it to be `Bool`. Thus, it is natural to assign $\star$ to y. However, this is undesirable since we would treat the application `succ_or_not True 'a'` as well typed, while its evaluation always leads to a cast error because no branch accepts a `Char` value.

To combat the inadequacy of $\star$, we assign to y a flow variation $F\langle \texttt{Int}, \texttt{Bool}\rangle$, which communicates that the type of y should be `Int` if the control flow enters the then branch and `Bool` otherwise. This now allows us to detect a cast error in `succ_or_not True 'a'`, since `Char` matches neither of the alternatives of $F\langle \texttt{Int}, \texttt{Bool}\rangle$. Alternatively, we could use recent work to assign gradual intersection or union types for such cases [Castagna and Lanvin 2017; Toro and Tanter 2017], but F variations give more precise results as shown via an example in Section 8.1.

The essence of typing conditionals is thus allowing variables to have different types in the branches. To support this, we need to use separate type environments and typing patterns for typing the branches, whose types may also differ. The rule If formally describes this idea. After typing the branches, the environments and patterns for branches are combined to type the entire condition

via $F\langle\Gamma_1,\Gamma_2\rangle$ and $F\langle\pi_1,\pi_2\rangle$. The environment $F\langle\Gamma_1,\Gamma_2\rangle$ essentially maps variables used with different types in Γ_1 and Γ_2 to variational flow types. For example, $F\langle\{y\mapsto \texttt{Int}\},\{y\mapsto \texttt{Bool}\}\rangle=\{y\mapsto F\langle\texttt{Int},\texttt{Bool}\rangle\}$. If a variable appears in only one environment, then it can be treated as having any type in the other environment. We also keep track of which conditionals introduce certain flow variations, by having $\ell\mapsto F_i\langle E_2,E_3\rangle$ in Ω. This is important when fixing inconsistencies in certain conditional branches. Overall, we assign the type $D_1\langle\star,\texttt{Bool}\rangle\to D_2\langle\star,F_1\langle\texttt{Int},\texttt{Bool}\rangle\rangle\to F_1\langle\texttt{Int},\texttt{Bool}\rangle$ to `succ_or_not`, where F_1 is the fresh variation for the conditional in `succ_or_not`. We give the detailed typing derivation for this example in Appendix A of the long version of this paper.[4]

Finally, the domain O is for reasoning about values produced by statically un-checkable expressions, determining if these values are used consistently when flowing into analyzable contexts. Consequently, the right alternative of each opaque variation will contain the type the opaque expression is consistently used with. To illustrate, consider the following Python program.

```
x = eval(input("Please enter an Integer: "))
x+1
```

We observe that `x` is used with type `Int` consistently in the program, but we cannot statically verify that `x` receives only an Integer at runtime (the user might provide `True` as the input). As a result, having the derivation $\top;\Gamma\vdash \texttt{x}:\texttt{Int}\mid\Omega$ is incorrect for the variable `x` in the above program, because $\top$ can only be attached to judgments that can be determined as valid at compile time. Meanwhile, we should not use the judgment $\bot;\Gamma\vdash \texttt{x}:\texttt{Int}\mid\Omega$ because $\bot$ is attached to judgments that can be determined as invalid at compile time.

We introduce $\dagger$, a new pattern construct, for such situations. This pattern can be attached to judgments that can be statically determined to be valid but at runtime the values may violate assumed types, when, for example, such values depend on user inputs. With the $\dagger$ pattern, we can now assign `x` the opaque variational type $O\langle\star,\texttt{Int}\rangle$, and the derivation $O\langle\top,\dagger\rangle;\Gamma\vdash \texttt{x}:O\langle\star,\texttt{Int}\rangle\mid\Omega$ indicates that the expected type at runtime for `x` is `Int`. Overall, Opaque in Figure 4 captures this intuition of trying to find an alternative static type for normally uncheckable expressions, where the $\dagger$ pattern indicates that the alternative static type is an expectation but may not hold at runtime. We add $\ell\mapsto O_i\langle\star,V\rangle$ to Ω to record which opaque expressions introduce their opaque variations.

4.3 Other Typing Rules

Now that we have already discussed the introduction of different variational domains, we turn to the remaining typing rules, which appear in Figure 5. The rule for constants (Con) is standard. A variable reference (Var) simply looks up the type in the environment and instantiates the type scheme over flow variations. We use $\overline{F}$ to denote a sequence of flow variations. We discuss the necessity of flow variation instantiation when we discuss the rule for typing **let** expressions.

There is an interesting interaction between F variations and **let** expressions. To illustrate, recall the `succ_or_not` function and consider the pair: (`succ_or_not True` 1, `succ_or_not False True`). Given that the type for `succ_or_not` is $D_1\langle\star,\texttt{Bool}\rangle\to D_2\langle\star,F_1\langle\texttt{Int},\texttt{Bool}\rangle\rangle\to F_1\langle\texttt{Int},\texttt{Bool}\rangle$, the two calls generate the typing patterns $F_1\langle\top,\bot\rangle$ and $F_1\langle\bot,\top\rangle$, respectively. As a result, the best typing pattern we can assign to this expression is $F_1\langle\bot,\bot\rangle$, which is equivalent to $\bot$, the same as saying this expression contain runtime inconsistencies. In fact, however, no such inconsistencies are present. The problem is that while different calls of `succ_or_not` will likely take different branches or `succ_or_not`, the types and patterns fail to reflect this fact.

To alleviate this problem, we introduce *choice type schemes* (quantifying over flow choices) so that the two references to `succ_or_not` generate the separate types $D_1\langle\star,\texttt{Bool}\rangle\to D_2\langle\star,F_2\langle\texttt{Int},\texttt{Bool}\rangle\rangle\to F_2\langle\texttt{Int},\texttt{Bool}\rangle$ and $D_1\langle\star,\texttt{Bool}\rangle\to D_2\langle\star,F_3\langle\texttt{Int},\texttt{Bool}\rangle\rangle\to$

[4] https://people.cmix.louisiana.edu/schen//ws/techreport/uslong.pdf

$$\textsc{Con}\ \frac{c \text{ is of type } \gamma}{\top;\Gamma \vdash c : \gamma \mid \varnothing} \qquad \textsc{Var}\ \frac{x \mapsto \forall \overline{F}.E_1 \in \Gamma \qquad \overline{F_i}\ \textit{fresh} \qquad E = \{\overline{F} \mapsto \overline{F_i}\}(E_1)}{\top;\Gamma \vdash x : E \mid \varnothing}$$

$$\textsc{Let}\ \frac{\pi;\Gamma, x \mapsto E_1 \vdash e_1 : E_1 \mid \Omega_1 \qquad \overline{F} = FC(E_1) - FC(\Gamma) \qquad \pi;\Gamma, x \mapsto \forall \overline{F}.E_1 \vdash e_2 : E_2 \mid \Omega_2}{\pi;\Gamma \vdash \textbf{let}\ x = e_1\ \textbf{in}\ e_2 : E_2 \mid \Omega_1 \cup \Omega_2}$$

$$\textsc{App}\ \frac{\pi;\Gamma \vdash e_1 : E_1 \mid \Omega_1 \qquad \pi;\Gamma \vdash e_2 : E_2 \mid \Omega_2 \qquad dom_\pi(E_1) \approx_\pi E_2}{\pi;\Gamma \vdash e_1\ e_2 : cod_\pi(E_1) \mid \Omega_1 \cup \Omega_2}$$

$$\textsc{Weaken}\ \frac{\pi;\Gamma \vdash e : E \mid \Omega \qquad \pi_1 \leq \pi}{\pi_1;\Gamma \vdash e : E \mid \Omega}$$

$$\begin{array}{lll} dom(\star) & = \star & \qquad cod(\star) & = \star \\ dom(E_1 \to E_2) & = E_1 & \qquad cod(E_1 \to E_2) & = E_2 \\ dom(d\langle E_1, E_2\rangle) & = d\langle dom(E_1), dom(E_2)\rangle & \qquad cod(d\langle E_1, E_2\rangle) & = d\langle cod(E_1), cod(E_2)\rangle \end{array}$$

Fig. 5. The remaining typing rules. For cases not listed, *dom* and *cod* are undefined.

$F_3\langle \texttt{Int}, \texttt{Bool}\rangle$, which have distinct F variation indices. Consequently, the patterns for these two calls become $F_2\langle \top, \bot\rangle$ and $F_3\langle \bot, \top\rangle$ and the best pattern for this expression becomes $F_2\langle F_3\langle \bot, \top\rangle, \bot\rangle$, which indicates that the pair can potentially run without failures, due to the presence of $\top$. We realize this idea by introducing choice schemes using Let for **let** expressions and instantiating them using Var for variable references. The structure for Let is standard except for the choice scheme $\forall \overline{F}.E$ and the *free choices* function *FC*, which collects all F variations in the given type (or type environment). Note that we support recursive let-bindings by typing e_1 with x.

The rule for typing function applications (App) uses the pattern-constrained compatibility relation introduced in Section 2.3. We use the dom_π and cod_π functions to compute the domain and codomain of types that are compatible with function types. The subscript π indicates that these functions need to be defined only for the variants where π has a $\top$. For example, $dom_\top(\texttt{Int})$ is invalid since $dom(\texttt{Int})$ is not well-defined, but $dom_\bot(\texttt{Int})$ is valid since the $\bot$ indicates that this is untrustworthy.

The main goal of Weaken is to relax the validity requirement π in the sense that we can use another pattern π_1 that has $\bot$ in equal or more places. This typing rule relies on a *less-defined* relation $\leq$ from Chen et al. [2012a]. While the definition of π extends that of [Chen et al. 2012a] with a $+$, the $\leq$ stays the same because $+$ is more precise than $\bot$ and less precise than $\top$ and as before $\bot$ is less precise than $\top$. Essentially, $\pi_1 \leq \pi_2$ holds if for any variant where π_2 has a $\bot$ then π_1 also has a $\bot$. We give the definition of $\leq$ in Appendix B (of the long version of this paper).

The purpose of Weaken is that, when typing a compound expression (such as an application), we can use different patterns for typing subexpressions (the function and the argument) and use $\leq$ to adjust patterns for subexpressions to a same pattern for the compound expression to be typed.

4.4 Properties

In this section, we investigate the properties of the type system presented in Figures 4 and 5. Note that the type system essentially type checks a set of related programs, finds proper static types for dynamic parameters, replaces type annotations for static parameters, and explores the different uses of values produced by opaque expressions. For this reason, we should first ensure that the result of our exploratory typing is the same as individually typing each related program.

We start by discussing some assumptions and the method to obtain and type each individual program. We assume that after the typing $\pi; \Gamma \vdash e : E \mid \Omega$ completes, π, Ω, and E are simplified as much as possible in the sense that they do not contain variations with identical alternatives or variations with unreachable alternatives. For example, $d\langle \texttt{Int}, \texttt{Int} \rangle$ is simplified to `Int` and $d\langle \texttt{Int}, d\langle \texttt{Bool}, \texttt{String} \rangle \rangle$ is simplified to $d\langle \texttt{Int}, \texttt{String} \rangle$. An efficient simplification method was presented in Chen et al. [2014b].

As discussed in Section 4.1, once the typing $\pi; \Gamma \vdash e : E \mid \Omega$ finishes, Ω maps each parameter, opaque expression, and conditional to a variational type. The Ω thus records all potential ways of changing the original program. For example, Ω_{e3}, the Ω for typing the expression e3, is given in Section 4.1. In contrast, we use a program update K (the syntax is given in Figure 3) to characterize one way of changing the program. Each K can be obtained from the corresponding Ω by selecting it with a decision. We use δ^c to denote decisions that eliminate all D, S, and O variations. For example, for $\pi = S\langle D\langle \top, \bot \rangle, F\langle \bot, \top \rangle \rangle$, $\{S.1, D.2\}$ and $\{S.2\}$ are such decisions, while $\{S.1\}$ and $\{D.1\}$ are not. For a given δ^c and a Ω, we obtain a K through the function *update*, defined as follows. The function $exp2chc(\ell, \Omega)$ gives the name of the choice introduced at the location ℓ when typing the expression yields the exploratory mapping Ω. For example, $exp2chc(\ell_2, \Omega_{e3})$ is D_1 and $exp2chc(\ell_3, \Omega_{e3})$ is D_2 when typing the expression e3 in Figure 2.

$$K = update(\Omega, \delta^c) = \lfloor \Omega \rfloor_{\delta^c} = \{(\ell, \lfloor E \rfloor_{\delta^c}) \mid \ell \mapsto E \in \Omega \wedge exp2chc(\ell, \Omega).2 \in \delta^c\}$$

For example, let $\delta^c_{e3} = \{D_1.2, S_1.2, D_2.1\}$, we can derive the following K_{e3}.

$$K_{e3} = \lfloor \Omega_{e3} \rfloor_{\delta^c_{e3}} = \{\ell_1 \mapsto \texttt{Int}, \ell_2 \mapsto \texttt{Int}\}$$

When we apply K_{e3} to the expression e3, written as K_{e3}(e3), it changes the expression to $(\lambda^{\ell_1} y\colon \texttt{Int}.y)\ ((\lambda^{\ell_2} x\colon \texttt{Int}.x)\ ((\lambda^{\ell_3} z\colon \star.z)\ 42))$.

Given an expression e and a K, we use the judgment $K; \Gamma \vdash_G e : G$ to denote that e, under the environment Γ and configuration update K, has the type G. Intuitively, derivations using $\vdash_G$ are the same as in a typical gradual type system, but for any parameter at the location ℓ with $\ell \mapsto G \in K$, then the parameter type at the location ℓ is updated to have the type G. For example, we have $K_{e3}; \Gamma \vdash_G$ e3 : `Int`, which intuitively says that the expression e3 has the type `Int` if we update the parameter types at ℓ_1 and ℓ_2 to `Int`. We defer the formal presentation of $\vdash_G$ to Section **??**.

By relating the typing rules in Figure 5 to $\vdash_G$, the following theorem states that our type system is correct. We defer the proof of the theorem to Appendix C.

Theorem 4.1 (Exploratory typing correctness). *If $\pi; \Gamma \vdash e : E \mid \Omega$, then for any δ^c such that $\lfloor \pi \rfloor_{\delta^c} = \top$ we have $\lfloor \Omega \rfloor_{\delta^c}; \Gamma \vdash_G e : \lfloor E \rfloor_{\delta^c}$.*

We observe that typing rules in Figures 4 and 5 have some nondeterminism. For example, the V in Abs (through *varIntro*) can take any value, as can the π in all other rules. However, we can find a *best typing* for any expression e under any environment Γ, in the sense that the π is as good as possible and the Ω is as precise and general as possible. We make this idea formal through Theorem 4.2. The theorem employs a standard precision relation on gradual types (such as the one in Siek and Vachharajani [2008]) and we write $G_2 \sqsubseteq G_1$ when G_2 is more static than G_1. We extend this relation with an additional axiom $\alpha \sqsubseteq \gamma$ so that more general types are considered more precise (for example $\alpha \to \alpha \sqsubseteq \texttt{Int} \to \texttt{Int}$). We also write $K_2 \sqsubseteq K_1$ if K_1 and K_2 share the same domain and for all x in the domain $K_2(x) \sqsubseteq K_1(x)$.

Overall, Theorem 4.2 states that there is a most defined, most static, and most general typing derivation possible for any program, which can be computed through type inference (see below). The notion of best typing is critical in most of the lemmas and theorems in Section 5 that involve the machinery for repair steps S1-S3.

Theorem 4.2 (Best Typing). *For any e and Γ, there is a best typing $\pi; \Gamma \vdash e : E \mid \Omega$ such that for any $\pi_1; \Gamma \vdash e : E_1 \mid \Omega_1$, $\forall \delta^c . \lfloor \pi_1 \rfloor_{\delta^c} = \top \Rightarrow \lfloor \pi \rfloor_{\delta^c} = \top \wedge \lfloor E \rfloor_\delta \sqsubseteq \lfloor E_1 \rfloor_{\delta^c}$.*

Proof. The proof of this theorem is a consequence of the Lemmas **??** and **??** in Appendix C. □

4.5 Type Inference

Theorem 4.2 stated that for any expression and type environment, there exists a best typing. This best typing can be computed through a type inference algorithm. Since the type inference algorithm can be obtained by a relative simple extension of the variational type inference algorithm [Chen et al. 2012b] to handle dynamic types, we defer its presentation to Appendix D. The inference algorithm can also be understood as extending the inference algorithm of Garcia and Cimini [2015] with the support for variational types. An important difference with previous inference algorithms [Chen et al. 2012b; Garcia and Cimini 2015] is that conditional branches are allowed to have different types.

5 STEPS FOR REPAIRING INCONSISTENCIES

In this section, we focus on providing the repair steps S1-S3 outlined in Section 1. We use two examples throughout this section. The first example is the expression e3, for which the resulting pattern π_{e3} (reproduced from Section 3) and the exploratory map Ω_{e3} (reproduced from Section 4.4) are as follows.

$$\pi_{e3} = S_1\langle D_1\langle \top, \bot\rangle, \top\rangle \qquad \Omega_{e3} = \{\ell_2 \mapsto D_1\langle \star, \texttt{Int}\rangle, \ell_1 \mapsto S_1\langle \texttt{Bool}, \texttt{Int}\rangle, \ell_3 \mapsto D_2\langle \star, \texttt{Int}\rangle\}$$

The second expression, adapted from `succ_or_not` in Section 4.2, is given below.

$$\lambda^{\ell_1} x : \star . \lambda^{\ell_2} y : \texttt{Int} . \textbf{if}^{\ell_3}\ x\ \textbf{then}\ \texttt{succ}\ y\ \textbf{else}\ (\texttt{not}^{\ell_4} : \star)\ y \tag{e4}$$

The result pattern π_{e4} and the exploratory map Ω_{e4} for this expression can be computed using type inference (Section 4.5) and are given below.

$$\begin{aligned}\pi_{e4} &= F_1\langle \top, S_1\langle D_2\langle \top, \bot\rangle, \top\rangle\rangle \\ \Omega_{e4} &= \{\ell_1 \mapsto D_1\langle \star, \texttt{Bool}\rangle, \ell_2 \mapsto S_1\langle \texttt{Int}, F_1\langle \texttt{Int}, \texttt{Bool}\rangle\rangle, \ell_3 \mapsto F_1\langle \texttt{Int}, \texttt{Bool}\rangle, \\ &\quad\ \ell_4 \mapsto D_2\langle \star, \texttt{Bool} \rightarrow \texttt{Bool}\rangle\}\end{aligned}$$

To simplify the discussion in this section, we assume that the exploratory maps and result patterns do not contain O variations. We can extend our methods to support them by simply replacing O variations with their second alternatives, treating $\dagger$s as $\top$s, and decorating the results with "maybe".

To simplify our discussion below, we introduce some auxiliary functions and notions. We use $chcs(\delta)$ to collect all the choice names in δ, that is $chcs(\delta) = \{d \mid d.1 \in \delta \vee d.2 \in \delta\}$. We also use $chcs(\pi)$ to return the set of choices in π, for example, $chcs(D_1\langle S_1\langle \bot, \top\rangle, \top\rangle)$ yields $\{D_1, S_1\}$. We use π_F to denote a pattern that contains only $\bot$, $\top$, and F variations.

Finally, to formally talk about cast errors, we need an evaluation relation $e \Downarrow (v, \delta)$ that specifies that the cast-inserted expression of e reduces to the value v [Siek et al. 2015a]. We use `error` for v to denote that the evaluation of e leads to a cast error. Note that our `error` does not contain a blame label [Wadler and Findler 2009] because we are only focused on identifying whether a program fails, and not what it blames. The δ records the branches that were covered during the execution. For example, for e4, we have (e4 `True 3`) $\Downarrow$ $(4, \{F_1.1\})$, where F_1 is the variation for the conditional in e4 (Please refer to Ω_{e4} above). Similarly, (e4 `False 3`) $\Downarrow$ $(\texttt{error}, \{F_1.2\})$. The δ in the evaluation relation is mainly used in Theorems 5.2 through 5.4. The definition of $\Downarrow$ is given in Appendix F, which is a simple extension of standard gradual program reduction rules, like those given by Siek et al. [2015a]. We use $e \Uparrow$ to denote that evaluating e is divergent.

5.1 Detecting and Understanding Runtime Inconsistencies

Step S1 Given an expression e and its typing pattern π, this step detects at compile-time whether executing e will yield inconsistencies. We provide this functionality in two steps. First, from π, we generate a decision used to eliminate all choices in π. Intuitively, we take the second alternatives of all dynamic variations D_is (for removing type suppression) and the first alternatives of all static variations S_is (for testing e without changing its type annotations). Below, we define a function $descS1(\cdot)$ for this step.

$$descS1(\pi) = \{S_i.1 \mid S_i \in chcs(\pi)\} \cup \{D_i.2 \mid D_i \in chcs(\pi)\}$$

Second, we select π with the generated decision. If that leads to a $\bot$, then evaluating e *will* lead to a cast error. If the selection leads to a π_F variation, then the evaluation *may* lead to a cast error, depending on conditional branches that will be taken. For example, for π_{e3}, $descS1(\pi_{e3})$ = $\{D_1.2, S_1.1\}$. Since selecting π_{e3} with $\{D_1.2, S_1.1\}$ leads to a $\bot$, we can correctly derive that evaluating e3 yields a cast error. For π_{e4}, $descS1(\pi_{e4}) = \{S_1.1, D_2.2\}$. Selecting π_{e4} with that decision yields $F_1\langle\top, \bot\rangle$, a π_F variation. Thus, our conclusion is that evaluating the expression e4 may lead to a cast error (for example with the arguments `False` and `3`) or may not (for example with the arguments `True` and `3`). We can verify that this conclusion is correct.

Our detection result may contain false positiveness. To illustrate, consider the expression $(\lambda x\colon \star.(x\ \ 1, x\ \ \texttt{True}))\ \ \texttt{id}$. The typing pattern for this expression is $D_3\langle\top, \bot\rangle$, where D_3 is the fresh variation for the parameter x. Based on this pattern, we may derive that this expression contains an inconsistency. However, evaluation of this expression succeeds. The reason is that we do not infer higher-rank types [Peyton Jones et al. 2007] as alternative types.

We can eliminate such false positiveness through the idea of program updates introduced in Section 4.4. Specifically, if we have detected an inconsistency for a given expression and the inconsistency can be removed by changing some static type annotations of the expression, then the inconsistency is real. We formally capture this idea in the following theorem, where K_S updates only static type annotations.

Theorem 5.1 (Soundness of inconsistency detection). *Given e, let* $\pi; \varnothing \vdash e : E \mid \Omega$ *be the best typing. If* $\lfloor\pi\rfloor_{descS1(\pi)} = \bot$ *and there exists some* K_S *such that* $\pi_1; \varnothing \vdash K_S(e) : E_1 \mid \Omega_1$ *and* $\lfloor\pi_1\rfloor_{descS1(\pi_1)} = \top$*, then* $e \Downarrow (\texttt{error}, \delta)$.

In practice, we can determine if such a K_S exists by checking if a fix that changes only static type annotations exists, and a method for this is provided in Section 5.2.

The proof of this theorem is given in Appendix F.

Step S2 This step aims to find cloaking ⋆s (or cloaked parameters and expressions). Since each ⋆ corresponds to a unique D_i choice, we sometimes refer to this step as collecting cloaking variations. By definition, a ⋆ is cloaking if removing the type suppression of that ⋆ in the program makes the runtime inconsistency statically detectable by gradual type systems. Formally, let π be the simplified pattern for typing e and D_i is created for some ⋆, then D_i is cloaking if there is some δ such that $\lfloor\pi\rfloor_{\{D_i.1\}\cup\delta} = \top$ and $\lfloor\pi\rfloor_{\{D_i.2\}\cup\delta} = \bot$. For example, for the expression e3 and π_{e3}, we can choose δ= $\{S_1.1\}$ and $D_i = D_1$ to identify D_1 as a cloaking variation. We express this idea of collecting cloaking variations in the following function, which takes a typing pattern as the input.

$$cloakingVars(\pi) = \{D \mid \exists\delta.\lfloor\pi\rfloor_{\{D.1\}\cup\delta} = \top \wedge \lfloor\pi\rfloor_{\{D.2\}\cup\delta} = \bot\}$$

Collecting cloaking variations directly based on this definition has a high complexity because we need to consider all Ds in π and for each D we need to find a δ satisfying the conditions given in the definition. A more efficient way to find them is by checking if the pattern contains $D\langle\top, \bot\rangle$ as a smaller pattern for some D. For example, as $\pi_{e3} = S_1\langle D_1\langle\top, \bot\rangle, \top\rangle$ contains $D_1\langle\top, \bot\rangle$ as

a smaller pattern, we immediately conclude that D_1 is cloaking. We give an approach, together with an algorithm, for collecting cloaking variations use this idea in Appendix E. The algorithm is able to handle more complicated patterns that has nested variations such as $D_1\langle\top, D_2\langle\top, \bot\rangle\rangle$. The algorithm has a linear complexity to the size of the input pattern.

5.2 Fixing Wrong Type Annotations

Determine if cast errors can be fixed Our first step in computing fixes is to determine if the cast errors can be fixed solely by changing type annotations, as motivated in Section 1.1. Not every cast error can be fixed in this manner. For example, $\lambda^{\ell_1} x\colon \texttt{Int}.\texttt{succ}\ (\texttt{(not:}\star\texttt{)}\ x)$ is statically well typed but always leads to a cast error, no matter how we change x's type annotation. On the other hand, some programs have multiple fixes. For example, $(\lambda^{\ell_1} x\colon \texttt{Bool} \rightarrow \texttt{Bool}.x)\ (\lambda^{\ell_2} y\colon \star.y)\ (\lambda^{\ell_3} z\colon \texttt{Int}.z)$ has two fixes: changing $\texttt{Bool} \rightarrow \texttt{Bool}$ for x to $\texttt{Int} \rightarrow \texttt{Int}$ or $\texttt{Int}$ for z to $\texttt{Bool}$. This expression's typing pattern is $\pi_{mul} = D_1\langle\top, S_1\langle S_2\langle\bot, \top\rangle, \top\rangle\rangle$, assuming the variations created for ℓ_1, ℓ_2, and ℓ_3 are S_1, D_1, and S_2, respectively.

To collect all possible fixes, it is helpful to view a pattern as a tree, where leaves are $\top$s or $\bot$s and internal nodes are variation names. For π_{mul} from the previous paragraph, it has four leaves, which are $\top$, $\bot$, $\top$, and $\top$ from left to right. It has three internal nodes, D_1, S_1, and S_2. Each leaf corresponds to a program configuration, which can be determined by moving up the tree from that leaf to the root. For example, the leaf marked with a $\top$ at the second alternative of S_2 corresponds to the configuration that changes the type annotation for z (moving up the tree will take the second alternative of S_2), keeps the type annotation for x (the first alternative of S_1), and changes the type annotation for y (the second alternative of D_1).

Therefore, we collect all fixes by first collecting *good* configurations. A configuration is good if (1) the corresponding leaf is $\top$ and (2) the configuration does not cover the first alternative of any D_i variation since that may hide type inconsistencies. Based on this idea, we define the function *goodConfigs* for collecting all good configurations as follows.

$$
\begin{aligned}
\mathit{goodConfigs}(\top) &= \{\varnothing\} \quad \mathit{goodConfigs}(\bot) = \varnothing \\
\mathit{goodConfigs}(S_i\langle\pi_1, \pi_2\rangle) &= \{\{S_i.1\} \cup \delta \mid \delta \in \mathit{goodConfigs}(\pi_2)\} \\
&\quad \cup \{\{S_i.2\} \cup \delta \mid \delta \in \mathit{goodConfigs}(\pi_2)\} \\
\mathit{goodConfigs}(D_i\langle\pi_1, \pi_2\rangle) &= \{\{D_i.2\} \cup \delta \mid \delta \in \mathit{goodConfigs}(\pi_2)\} \\
\mathit{goodConfigs}(F_i\langle\pi_1, \pi_2\rangle) &= \mathit{goodConfigs}(\pi_1) \cup \mathit{goodConfigs}(\pi_2)
\end{aligned}
$$

The general idea of *goodConfigs* is that it starts from leaves and accumulates configurations as it moves up to the root, where *goodConfigs* terminates. For $\top$, *goodConfigs* returns a unit set whose element is an empty set. This allows *goodConfigs* to accumulate configurations in recursive calls. Instead, for $\bot$, it returns an empty set. For each internal node, it performs different operations. For a static variation S_i, it expands configurations from the left child with $S_i.1$ and those from the right child with $S_i.2$ and returns them. For a dynamic variation D_i, it discards configurations from the left child and expands those from the right with $D_i.2$ and returns them. For a flow variation F_i, it combines configurations from both children and returns them.

Since each good configuration corresponds to a fix, we can determine if the inconsistencies can be fixed as follows. Assume π is the pattern for typing e, then the inconsistencies in e cannot be fixed by changing type annotations if $\mathit{goodConfigs}(\pi)$ is empty. Otherwise, for any δ from $\mathit{goodConfigs}(\pi)$ $\lfloor\pi\rfloor_\delta = \top$ implies the the inconsistencies in e can be fixed and $\lfloor\pi\rfloor_\delta = \pi_F$ implies the inconsistencies can possibly be fixed, depending on the conditional branches being taken when evaluating e.

For the expression e3 and its pattern π_{e3}, $goodConfigs(\pi_{e3}) = \{\{S_1.2, D_1.2\}\}$ and selecting π_{e3} with the decision in the set yields a $\top$, the inconsistency can thus be fixed. For the expression e4 and its pattern π_{e4}, $goodConfigs(\pi_{e4}) = \{\{S_1.2\}\}$ and selecting π_{e4} with that decision leads to $F_1\langle\top, \top\rangle$, which is the same as $\top$, meaning that the inconsistency in the expression e4 can also be fixed. For the expression given in the first paragraph of this section and its pattern π_{mul}, $goodConfigs(\pi_{mul}) = \{\{S_2.2, S_1.1, D_1.2\}, \{S_1.2, D_1.2\}\}$. Selecting π_{mul} with either decision in the set yields a $\top$, and the inconsistency in it can thus be fixed.

This idea of determining whether inconsistencies can be fixed is correct, expressed in the following theorem.

Theorem 5.2. *Let $\pi; \varnothing \vdash e : E \mid \Omega$ be the best typing for e, $\pi_1 = \lfloor \pi \rfloor_{\delta_1}$ where $\delta_1 \in goodConfigs(\pi)$, and $e_\star$ is obtained by removing all type annotations in e, then*

- *$\pi_1 = \top$ and $e_\star \Downarrow (v, \delta)$ imply $v \neq$ `error` for some δ.*
- *$\pi_1 = \pi_F$, $e_\star \Downarrow (v, \delta)$, and $\lfloor \pi_F \rfloor_\delta = \top$ implies $v \neq$ `error`.*
- *$\pi_1 = \top$ may imply $e_\star \Uparrow$.*

This theorem is reminiscent of the dynamic part of gradual guarantee [Siek et al. 2015a]. Note, $\pi_1 = \top$ does not necessarily imply that $e_\star$ terminates. For example, consider the following expression. [5]

$$\textbf{let } f = \lambda^{\ell_1} x\colon \star. f\ (x+1) \textbf{ in } f\ ((\lambda^{\ell_2} y\colon \texttt{Bool}. y)\ (1^{\ell_3} : \star)) \tag{e5}$$

This expression is statically well typed but has a cast error. The reason is that the expression 1 that has runtime type `Int` is passed into a function whose parameter type is `Bool`. For this expression, the result pattern π_{e5} and the exploratory map Ω_{e5} are given as follows, where the variations for x, y, and the ascription are D_1, S_2, and D_3, respectively.

$$\pi_{e5} = S_2\langle D_3\langle D_1\langle \top, \bot\rangle, \bot\rangle, \top\rangle \qquad \Omega_{e5} = \{\ell_1 \mapsto D_1\langle \star, \texttt{Int}\rangle, \ell_2 \mapsto S_1\langle \texttt{Bool}, \texttt{Int}\rangle, \ell_3 \mapsto D_2\langle \star, \texttt{Int}\rangle\}$$

Based on the definition of *goodConfigs*, we have $goodConfigs(\pi_{e5}) = \{\{S_2.2\}\}$. Let $\delta^c_{e5} = \{S_2.2\}$, we have $\lfloor \pi_{e5} \rfloor_{\delta^c_{e5}} = \top$. However, running the expression that removes all type annotations from e5 is non-terminating, due to the recursive call inside f that has no base case. To our best knowledge, no work exists in gradual typing that reasons about program termination as type annotations are removed.

Compute fixes for inconsistencies From each good configuration, we can derive a fix. Based on the definition *goodConfigs*, each good configuration contains some selectors of the form $S_i.2$ if the program contains inconsistencies. As S_i variations are created for parameters with static type annotations (please refer to Section 3 for an informal example and Section 4.2 for formal rules). Moreover, the first and second alternatives of such variations are the original type annotation and the type annotation it should be changed to remove the inconsistencies. Consequently, the fix corresponds to each good configuration needs to update the type annotations for parameters where S_is are introduced.

A fix is thus a program update, and we reuse the idea of generating program updates in Section 4.4 to produce a fix. Specifically, we implement this idea in the following functions, with $\pi; \Gamma \vdash e : E \mid \Omega$ being the best typing for e and δ is from $goodConfigs(\pi)$,

$$fix(e, \delta) = update(\Omega, chgPars(\delta)) \qquad chgPars(\delta) = \{S_i.2 \mid S_i.2 \in \delta\}$$

The *update* function is defined in Section 4.4. The function $chgPars(\delta)$ removes all $D_i.2$s from δ because we do not change $\star$s for fixing inconsistencies. Based on *chgPars*, we can find the minimal

[5] This expression uses ascription to specify that 1 has the type $\star$ at compile time. While our expression syntax does not support ascription, (1:$\star$) could be easily represented in our syntax as $(\lambda z\colon \star. z)\ 1$.

fix by going through all good configurations and find the set that changes the fewest parameters. We define a function $minalFix(e)$ for computing the minimal fix for e. The definition, though space consuming, is simple; we omit its definition here.

For the expression e3 and its pattern π_{e3} and exploratory map Ω_{e3}, we have:

$$minalFix(\text{e3}) = update(\Omega_{e3}, chgPars(\{S_1.2, D_1.2\})) = update(\Omega_{e3}, \{S_1.2\}) = \{\ell_2 \mapsto \texttt{Int}\}$$

Thus, the sole (and minimal) fix for e3 is changing its annotation for the parameter y to `Int`.

For the expression e4 and its π_{e4} and Ω_{e4}, we have:

$$minalFix(\text{e4}) = update(\Omega_{e4}, chgPars(\{S_1.2\})) = update(\Omega_{e4}, \{S_1.2\}) = \{\ell_2 \mapsto F_1\langle \texttt{Int}, \texttt{Bool}\rangle\}$$

Thus, the only fix for e4 is changing its annotation for the parameter y to a flow type. However, since type checkers and users may not be able to make use of variational types directly, we will have to suggest changing the type annotation of y to a $\star$.

In addition, if the computed fix suggests changing some parameter type to a type variable but the language does not support type variables as annotations, we must also resort to $\star$. One such example is the fix for `asciify` in Section 1.

Our computed fix is correct in the sense that the fixed expression will no longer produce cast errors, as captured in the following theorem, where $K(e)$ applies the update K to e (Section 4.4).

Theorem 5.3. *Let $\pi; \varnothing \vdash e : E \mid \Omega$ be the best typing for e, δ is any member of $goodConfigs(\pi)$, and $K = fix(e, \delta)$, then*

- *$K(e) \Downarrow (v, \delta_1)$ such that $v \neq \texttt{error}$ for some δ_1, or*
- *$K(e) \Uparrow$.*

Similar to Theorem 5.2, $K(e)$ may be divergent. The expression e5 is one such example. For e5, we have $K_{e5} = fix(e5, \delta^c_{e5}) = \{\ell_2 \mapsto \texttt{Int}\}$, and $K_{e5}(e5)$ changes the type for y to `Int`. It is easy to check that the resulting expression is indeed non-terminating.

For a given expression e, our fix not only removes the cast error, but is "meaningful" in the sense that the expression obtained by applying the fix to e and the expression that removes all type annotations in e (we call it $e_\star$ in the theorem) evaluate to the same value. The following theorem expresses this connection.

Theorem 5.4. *Let $\pi; \varnothing \vdash e : E \mid \Omega$ be the best typing for e, δ is any member of $goodConfigs(\pi)$, and $K = fix(e, \delta)$, then*

- *$e_\star \Downarrow (v_2, \delta_2)$ implies $K(e) \Downarrow (v_1, \delta_1)$ such that $v_1 = v_2$ and $\delta_1 = \delta_2$.*
- *$e_\star \Uparrow$ implies $K(e) \Uparrow$.*

Again, we observe that both the dynamic version (removed all type annotations) and the fixed version (applied K_{e5}) of e5 are non-terminating.

6 EXTENSIONS

As our prototype and evaluation are for Python, it would be useful to show how to extend our machinery so far to handle common features found in Python. We particularly consider objects and structural subtyping. Our extension is built on top of the work by Siek and Taha, who studied combining gradual typing and objects [2007].

To support objects, we first need to extend our type syntax to represent objects, which is a list of label and type pairs [Siek and Taha 2007]. For example, the object type $[l_1 : \texttt{Int}, m_1 : \texttt{Int} \rightarrow \star]$ includes a filed l_1 (we use ℓs to denote program locations and ls to denote field labels.) whose type is `Int` and a field m_1 whose type is $\texttt{Int} \rightarrow \star$. We skip the extension of type syntax here since it is rather straightforward and it would take quite much space.

The syntax used for creating objects in Siek and Taha [2007] is $[l_i = G_{ir}\ \varsigma(x_i : G_{ig})e_i\ ^{\forall i \in 1 \ldots n}]$, where l_i is the label and ς indicates a method definition (just like λ indicates a function definition). For a method, x_i denotes the parameter, e_i denotes the body, G_{ig} denotes the parameter type, and G_{ir} denotes the return type. Our rule for typing objects is given below.

$$\textsc{Obj}\ \frac{\begin{array}{c} E_{ig} = \mathit{varIntro}(G_{ig}) \qquad \pi; \Gamma, \texttt{self} \mapsto E_\rho, x_i \mapsto E_{ig} \vdash e_i : E_{ir} \mid \Omega_i \\ E_{ir}|_1 = G_{ir} \qquad \forall i \in 1 \ldots n \qquad E_\rho = [l_i : E_{ig} \to E_{ir}\ ^{\forall i \in 1 \ldots n}] \end{array}}{\pi; \Gamma \vdash [l_i = G_{ir}\ \varsigma(x_i : G_{ig})e_i\ ^{\forall i \in 1 \ldots n}] : E_\rho \mid \Omega_i \cup \ldots \cup \Omega_n}$$

This rule is very similar to the rule for typing objects in Siek and Taha [2007] with two main differences. The first difference is that, in our rule, for each parameter type G_{ig} we construct E_{ig} by introducing variations through *varIntro*, as we did for typing abstractions in Section 4.2. We then type the body e_i under the assumption that x_i has the type E_{ig}. The second difference is that in Siek and Taha [2007] the body e_i should have the specified type G_{ir}. In our rule, however, since we have changed the parameter type from G_{ig} to E_{ig}, we should not make the same requirement. Instead, we denote the body type as E_{ir} and require that the type yielded from taking the first alternatives of all variations in E_{ir} (denoted as $E_{ir}|_1$ in the third premise of the rule) should be the same as G_{ir}. As an example of $\cdot|_1$, we have $D_1\langle S_2\langle \texttt{Int}, \texttt{Bool}\rangle, \texttt{Bool}\rangle|_1 = \texttt{Int}$.

We next move to type method invocation. In Siek and Taha [2007], for an invocation $e_1.l(e_2)$ to be well typed, the type of the argument (G_2) and the parameter type (G_1) should satisfy the relation $G_2 \lesssim G_1$. Intuitively, $G_2 \lesssim G_1$ if all parts that are static in both G_2 and G_1 satisfy the subtyping relation. For example, $[l : \texttt{Int} \to \star] \lesssim [l : \texttt{Int} \to \texttt{Bool}]$ but $[l : \texttt{Int} \to \star] \not\lesssim [l : \texttt{Bool} \to \texttt{Bool}]$.

We extend $\lesssim$ to handle variations and allow it to hold in certain alternatives only through the relation $\lesssim_\pi$, defined as follows.

$$\frac{\forall \delta . \lfloor \pi \rfloor_\delta = \top \Rightarrow \lfloor E_2 \rfloor_\delta \lesssim \lfloor E_1 \rfloor_\delta}{E_2 \lesssim_\pi E_1}$$

Intuitively, for $E_2 \lesssim_\pi E_1$ to hold, we require only the alternatives that π have $\top$ the $E_2 \lesssim E_1$ holds and disregard the $\lesssim$ between E_2 and E_1 at other alternatives. As an example of $\lesssim_\pi$, we have $[l : \texttt{Int} \to \star] \lesssim_{S_1\langle \top, \bot\rangle} [l : S_1\langle \texttt{Int}, \texttt{Bool}\rangle \to \texttt{Bool}]$.

With $\lesssim_\pi$, we could type method invocation as follows.

$$\textsc{Invk}\ \frac{\pi; \Gamma \vdash e_1 : [\ldots, l : E_1 \to E, \ldots] \mid \Omega_1 \qquad \pi; \Gamma \vdash e_2 : E_2 \mid \Omega_2 \qquad E_2 \lesssim_\pi E_1}{\pi; \Gamma \vdash e_1.l(e_2) : E \mid \Omega_1 \cup \Omega_2}$$

The only difference between this rule and the one in Siek and Taha [2007] is that here we use $\lesssim_\pi$ rather than $\lesssim$ in theirs. As the rule is quite self-explanatory, we will not elaborate further.

All other typing rules in Siek and Taha [2007] are simpler than the object creation and method invocation rules we have presented. Extending these rules to deal with variations is straightforward, and we omit it here.

Another interesting language feature to consider here is variable assignment, such as $x := e$ that assigns e to x. The work on Flow [Chaudhuri et al. 2017] has developed a nice solution for this. The idea is to extend the typing judgment to pass out a type environment that records changes made to it. We can literally reuse their rule here for this purpose, and we will not repeat it here.

7 EVALUATION

To evaluate the feasibility of exploratory typing, we have implemented the ideas developed in this paper in PyHound, a tool (built on the guarded variant of Reticulated 0.1) for detecting and fixing inconsistencies in Reticulated Python [Vitousek et al. 2017] programs. We support

structural subtyping by inferring the least required fields of a parameter based on the function body. We evaluate PyHound's effectiveness (i.e. its ability to find and fix errors) and performance in Sections 7.1 and 7.2, respectively.

7.1 PyHound Effectiveness

In this section, we evaluate PyHound's ability to support each of the repair steps S1-S3.

Evaluation setup Ideally, we could test PyHound on an existing benchmark with different inconsistencies. However, we are not aware of any such program sets. Thus, to make our evaluation result meaningful, we drew evaluation programs from three different sources: (1) We adopted programs from the artifact created by Campora et al. [2018a] (which adapted programs from the Python benchmark suite on github) and adapted some programs from Rosetta Code's Python entries, totaling 8 programs. To make our test results representative, each original program we considered already contained about 20% to 70% of parameters that had static type annotations. (2) We synthesized two large programs by merging the 8 programs from source (1) repeatedly, making necessary function renaming to avoid name conflicts. These two programs are to stress test PyHound. (3) We took all programs that have cast errors from student logging. Gradual typing was included as an optional topic in an undergraduate programming language course and a graduate course one coauthor taught. We have created a web interface running Reticulated Python to allow students to migrate the 8 benchmark programs from the source (1) as well as migrate their own programs. We have 82 programs that have cast errors, and they contain 117 cast errors in total. Their sizes range from 69 to 277 LOC.

For each of programs from sources (1) and (2), we randomly added type annotations to function parameters with a Bash script until an inconsistency was added. The inserted type annotations were constructed by binary type constructors (including $\rightarrow$ for constructing function types and (,) for constructing tuple types in Python), unary type constructors (including `List`), and other primitive types (including `Int Bool`, etc.). For example, the types that were inserted include $\star \rightarrow$ `Int`, `List(`$\star$`)`, ($\star$,$\star$), and so on.

We repeated this process to generate 20 unique configurations of each program, with each containing at least one dynamic inconsistency. Automatically generating a configuration with dynamic inconsistencies is tricky because randomly inserting type annotations quickly creates static inconsistencies that we need to discard. It was common that our script needed to run for hours of generating and discarding programs before encountering a program with dynamic inconsistencies. For this reason, we considered only 20 programs with dynamic inconsistencies at different locations for each benchmark. Overall, this leads to 200 distinct programs containing inconsistencies based on sources (1) and (2). Figure 6 lists the details of the programs and their metrics.

Evaluation results We now discuss how PyHound supports each repair step. To decide if PyHound supports step S1 on each program, we check the result reported by PyHound against running the program. If they agree, for example, both indicated a cast error, then the result by PyHound is correct. From Figure 6, we observe that PyHound is able to detect inconsistencies in 196 out of 200 programs (with a ratio of 98%). It did not detect errors in 4 variant programs of the runge benchmark. All of these errors have the form in Figure 7 (left). PyHound fails to detect an inconsistency in the snippet because PyHound and Reticulated Python do not type-check library functions and thus they have type $\star$. Since all of `sqrt`, `map`, `enumerate`, and `range` have the type $\star$, `y` interacts with no statically typed code and thus no type information can be exploited by PyHound to detect the inconsistency caused by the incorrect annotation for `y`.

We have also evaluated the effectiveness of pytype [2020] on detecting inconsistencies (S1) using the same protocol as we evaluated PyHound (We have to make necessary type annotation changes

Name	LOC	#Par	#Err	S1		S2	S3	Time	
				PyHound	pytype			PyHound	pytype
meteor	238	28	24	20	3	20	20	0.16	3.37
nbody	164	17	20	20	5	20	20	0.08	3.25
pascal	68	9	32	20	12	32	32	0.04	1.74
raytrace	277	68	26	20	5	26	26	0.47	3.28
runge	74	11	24	16	6	16	16	0.07	2.00
sci_mark	221	23	36	20	3	32	32	0.25	2.79
spectral	69	9	20	20	5	20	20	0.04	2.44
tic_tac	87	5	28	20	14	24	24	0.05	2.19
syn_1	4521	951	24	20	9	24	24	16.85	47.85
syn_2	15255	2451	24	20	4	24	24	97.08	550.94
Overall	-	-	258	196/200	66/200	238/258	238/258	-	-

Fig. 6. Effectiveness evaluation of PyHound and pytype. The first three columns give the name, the size in LOC, and the number of parameters in the program. The #Err column gives the total number of inconsistencies within all 20 generated programs. The S1 through S3 columns show the effectiveness of PyHound and pytype (which supports S1 only) supporting each repair step. The Time column measures the average time for PyHound and pytype to type check and generate suggestions across the 20 programs. All times in this paper are in seconds and are measured on a System76 Galago Pro with a Intel Core i5-8250U CPU @ 1.60GHz, running 64 bit Ubuntu 16.04 LTS. Each time is an average of 10 runs.

```
def f(x,y:Bool):
  return x * sqrt(y)
l = enumerate(range(20))
map(f, l)
```

```
def square(x, y:String):
  return x ** 2
square(4, sqrt(16))
```

Fig. 7. Code adapted from benchmarks, that pose difficulties to PyHound on S1 and S2 (left) and S3 (right).

for pytype to work, for example, changing `Dyn` to `Any`.). Pytype is probably the most powerful Python tool to detect inconsistencies between user type annotations and the code and is used by thousands of projects at Google. The result (in Figure 6) indicates that pytype could detecting inconsistencies in 33% of programs, much lower than the ratio of PyHound.

For all the 82 student programs containing cast errors, PyHound is able to detect inconsistencies in 78 programs (yielding a ratio of 95%) while pytype could detect only 30 (with a ratio of 37%) of them. Cast errors in 4 student programs were not detected by PyHound are due to the interactions of library functions, very similar to the situation given in Figure 7. This shows that PyHound is more effective at detecting inconsistencies in student programs.

To determine if PyHound supports S2, we added a type annotation to each variable whose ⋆ PyHound identifies as cloaking. We determine that PyHound has correctly identified a cloaking ⋆ if the added type annotation caused a static type error. In all the 258 cast errors, PyHound is able to correctly identify cloaking ⋆s for 238 errors. PyHound misses other cases because these inconsistencies are not detected, due to the same reason for S1. The difference for the results between S1 and S2 means that while some single programs contain multiple inconsistencies, PyHound detects some, but not all inconsistencies.

To evaluate S3, we follow the fix messages from PyHound by replacing the static type annotations with the suggested types. If this removes the inconsistency, then PyHound supports S3 for the given program. Otherwise, it does not. From Figure 6, we see that PyHound removes 100% of all

inconsistencies across all the generated programs for six programs. In average, it removes 93% of all inconsistencies.

The only inconsistencies that PyHound fails to compute fixes for, are those that cannot be detected. Such errors are caused because: (1) the parameter is used only with type ⋆ in a function body (or is unused) and (2) the arguments flowing into this parameter all have type ⋆. One such example, shortened from the meteor program, is presented in Figure 7 (right). Since `y` is not used in `square` and only received an input from `sqrt(16)`, which has type ⋆ due to the way Reticulated Python handles library functions. Essentially, the ubiquitous interaction of dynamic types with `y` hides the useful type information for removing the incorrect type annotation.

For all the 117 inconsistencies in the student programs, PyHound is able to find correct cloaking ⋆s and propose fixes to 108 inconsistencies, with a correct ratio of 92%.

Overall, we conclude that PyHound is effective in helping users detect (S1), understand (S2), and fix (S3) inconsistencies, with ratios of 98%, 93%, and 93%, respectively.

Discussions Our evaluation results on the datasets demonstrate that PyHound is effective. We next investigate the generalizability of our results. Our investigation covers the following aspects.

We first investigated the amount of type information that could be inferred. This information is important because all three reparation steps of PyHound rely on inferred types. For example, to detect inconsistencies, we need to infer types for parameters with ⋆s. To compute fixes, we need to find correct types for parameters with static type annotations. However, type inference for Python is tricky. On the one hand, Python is dynamically typed, and, like other such languages, many language features and expressions in programs can not be captured using static types and thus by type inference. On the other hand, as observed by Takikawa et al. [2016], "static type systems accommodate common untyped programming idioms", implying that quite much type information may be inferred for dynamic programs.

Across different benchmarks, the amount of type information that could be inferred by PyHound varies significantly. Besides the initially given 20% to 70% of static annotations (detailed in our evaluation setup), the ratios of the parameters whose full static types could be inferred are about 33% for nbody, 42% for meteor, 50% for spectral and runge, 65% for tic_tac and pascal, 75% for sci_mark, and 82% for raytrace. For student programs, the inferred type ratio varies from 30% to 85%. In addition, in many cases, our approach infers types that are not fully static. For example, we may infer `List[⋆]`, (⋆,⋆), ⋆ → `Int`, etc.

Although the amount of type information that could be inferred for different programs are quite different, we observed that PyHound is quite consistent on detecting cast errors and finding fixes. Our deep investigation into this revealed two insights. First, while inferring types for the whole programs is difficult (due to the dynamic nature of Python), there are always regions whose types could be inferred. Interestingly. these are the places where static types could be added, where types could be inferred, and where cast errors are most likely to happen. For this reason, our approach is likely to detect these cast errors while full type inference is difficult. Second, while we could infer only parts of static types for some parameters, they are sufficient to detect cast errors. For example, if an inferred type is (⋆,`Int`) and a type (`String`,`Bool`) is given, then an inconsistency between `Int` and `Bool` could already be detected.

We then looked into the language features used by the evaluated datasets. We found out that a wide range of language features were covered by these datasets, including mutability, classes, objects, most control structures (conditionals, `while`, `for`, `break`, and `continue`), (mutual) recursive definitions, and so on. Finally, we have looked into the dependency graphs of our evaluated programs and the relation between dependency graph complexity and whether cast errors could be detected. We observed some correlation between them. Specifically, programs with more complex

dependency graphs, using more higher-order functions, and calling more library functions making cast error detection more difficult. However, complex dependency graphs or higher-order functions alone does not cause any difficulty.

Overall, PyHound is effective at detecting cast errors and suggesting fixes for programs that could be inferred with different amount of type information, that cover diverse language features, and that with different dependency graph complexities. We thus are quite confident that our evaluation results generalize to other programs.

7.2 PyHound Performance

Figure 6 presents the time duration of PyHound for all steps S1 to S3. To give a sense about the efficiency of PyHound, we have measured the times for Reticulated Python [Vitousek et al. 2017] to type check syn_1 and syn_2. These times are 3 and 13 seconds, respectively. These results indicate that PyHound incurs a very minor slowdown, within a factor of 8. Without PyHound, it would take $2^{2451} \times 13$ seconds (obtained by multiplying the brute-force search space and the time to handle a single configuration) to find fixes for syn_2. The figure also presents the time duration of pytype on the benchmarks. We observe that PyHound runs much faster than pytype does.

To more thoroughly test the scalability of PyHound as program size increases, we created large programs by repeating function definitions from the sci_mark and raytrace benchmarks in 1000 LOC increments from (≈1000 LOC) to (≈10,000 LOC). We present the evaluation result for these programs in Figure 8. Across the ten measured programs, the time growth is linear. Even for a program with 1,971 parameters (which creates 1,971 *D* and *S* variations), the overhead is small.

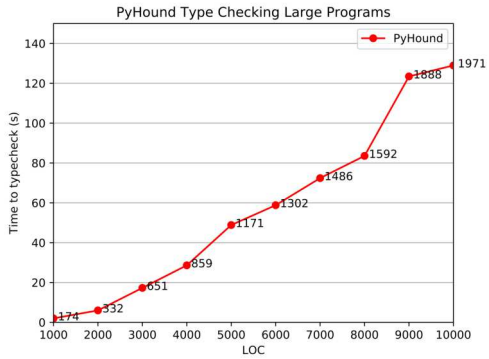

Fig. 8. PyHound performance with respect to increased program sizes. The numbers next to the points are the number of parameters.

Note, the difference between running synthetic, large programs and running small program multiple times is significant. The reason is that larger programs will have more functions and thus more parameters, more statically untypable code, and more conditionals, such programs will have more variations created during analysis. Synthetic programs thus test how well our approach handles a large number of variations. Based on Figure 8, running small programs multiple times needs to handle 174 variations at a time, whereas analyzing the synthesized program needs to handle as many as 1971 variations at once, a much more difficult task. Overall, we conclude that PyHound scales to large programs.

8 RELATED WORK

In this section, we discuss the relation of exploratory typing and repairing inconsistencies to related research directions in gradual typing and static contract analyses.

8.1 Type System Design

Designing more expressive gradual type systems has been an active area of research [Ahmed et al. 2011, 2017; Igarashi et al. 2017b,a; Jafery and Dunfield 2017; Kent et al. 2016; Lehmann and Tanter 2017; Siek and Taha 2007]. One popular type system extension present in many gradually typed languages is union types, which appear in Typed Racket [Tobin-Hochstadt and Felleisen 2008],

TypeScript, and JavaScript (Flow) [Chaudhuri et al. 2017]. Our flow variations provide similar functionality to the union types in these languages in that they similarly allow flow sensitive reasoning about variable use in different branches of conditional expressions. However, these union types differ from our flow variations in that they use explicit typecase expressions to handle values for the different types in the overall union. The typecase expression verifies which type the variable has before it is used with expressions expecting that type in the branch. In contrast, our approach does not require that a value's type is checked before uses in a branch.

Our flow variations bear greater resemblance to some more recent formalizations, namely gradual set theoretic types [Castagna and Lanvin 2017] and gradual union types [Toro and Tanter 2017]. Like our flow variations, both of these approaches forgo typecase expressions. A main difference with these formalizations is that our type representations are more precise about where inconsistencies can occur. Recall the `succ_or_not` function from Section 4.2, which can also be typed in both of these systems. They deem the expression `succ_or_not True True` as well typed and assign `Bool` to it, while in fact evaluating the expression leads to a cast error. In contrast, our type system assigns $F\langle \texttt{Int}, \texttt{Bool} \rangle$ to it with the pattern $F\langle \bot, \top \rangle$, which indicates that executing this expression may result in an inconsistency and must result in one when control flow enters the then-branch in `succ_or_not`, a more precise result for the purpose of detecting runtime inconsistencies.

8.2 Migrating Gradual Types

Campora et al. [2018b] developed a method called *Migrating Gradual Types* (MGT) that adds static types to as many parameters with ⋆s as possible. MGT suggests a static type for a ⋆ as long as using that static type will not cause static type errors. As a result, a migration by MGT may introduce cast errors to a given program. For example, for the following expression (reproduced from Section 3 for readability purposes), MGT suggests two possible migrations: 1) using `Bool` as the static type annotation for x and keep ⋆ for z or 2) use `Int` as the type annotation for z and keep ⋆ for x. We can verify that following these migrations the expression e3 remains statically well typed. However, the migration 1) introduces another cast error because at runtime the expression $(\lambda^{\ell_3} z\colon \star.z)\ 42$ produces an `Int` value that will be passed to the function $(\lambda^{\ell_2} x\colon \texttt{Bool}.x)$ whose parameter type is `Bool`.

$$(\lambda^{\ell_1} y\colon \texttt{Bool}.y)\ ((\lambda^{\ell_2} x\colon \star.x)\ ((\lambda^{\ell_3} z\colon \star.z)\ 42)) \tag{e3}$$

The expression e3 illustrates that MGT may introduce cast errors to an expression that already has cast errors. The following expression indicates that MGT may introduce type errors to expressions that have no cast errors.

$$(\lambda^{\ell_1} x\colon \star.\textbf{if}^{\ell_2}\ \texttt{True}\ \textbf{then}\ 3\ \textbf{else}\ x)\ (\texttt{True} : \star) \tag{e6}$$

For expression e6, MGT suggests a migration of using `Int` for the parameter x. MGT finds this migration as it requires branches of a conditional to have the same type, making the type of `3` being propagated to x and then to the parameter. This migration, however, will cause a cast error because the value (`True`:⋆) that has the type `Bool` will be passed to a function whose parameter type is `Int`.

Our approach in this paper could detect and suggest fixes to cast errors if both e3 and e6 were migrated following MGT's migrations.

Technically, MGT also makes use of variations to efficiently compute migrations. However, compared to that work, this paper poses the following unique challenges. First, this work allows conditional branches to have different types while in MGT they must have the same type. Our solution is to introduce F variations to represent types for conditionals. While union and intersection types [Castagna and Lanvin 2017] could also be used for this purpose, we believe that F variations have two benefits. First, as discussed in Section 8.1, F variations yield more precise results for detecting inconsistencies. Second, using F variations simplifies our type system since the paper

already includes the machinery for handling variations. If we use union and intersection types, we need to investigate the interaction between variational types and these types.

The second challenge is that this paper needs to reason about the types of subexpressions whose type can not be statically determined. Our solution is using O variations and introducing a new pattern construct $\dagger$ (Section 4.2) to represent and reason about types of such subexpressions. The third challenge is that different call sites of a function with conditionals will be synchronized on the branch being taken although different call sites may take different branches. One such example was given in the second paragraph of Section 4.3. Our solution is introducing choice type schemes such that different call sites will be instantiated with different flow variation names (Section 4.3), allowing different calls of a single conditional to take different branches.

8.3 Gradual Program Analysis and Contract Verification

Several dynamic analysis [Kristensen and Møller 2017b; Williams et al. 2017] and static analysis [Feldthaus and Møller 2014; Kristensen and Møller 2017a] approaches have been developed to detect the inconsistencies between type signatures in TypeScript definition files and their corresponding Javascript libraries and client code. Dynamic analysis based approaches do not provide sufficient support for our problem domain, for the same reasons as given for the dynamic gradual guarantee and blame tracking (Section 1.1). Another important difference is that while the mentioned previous work mainly focuses on detecting errors, our work also supports understanding and fixing inconsistencies by changing type annotations.

While the main goal of soft contract verification (SCV) [Nguyen et al. 2014] is not detecting inconsistencies in gradual typing, it is capable of detecting contract violations in Racket programs at compile time. This bears some similarity to our goal of statically detecting inconsistencies in gradually typed programs. Nevertheless, SCV and our approach differ in several notable ways.

First, SCV assumes that contracts are always correct and never reports them as a possible cause for an error, while our approach detects and fixes errors in type annotations, which can likely be wrong when migrating programs towards using more static checking [Williams et al. 2017]. Second, after observing a failing contract, SCV short circuits its analysis and propagates the error outwards. This means that it will not detect all of the contract failures for certain programs. For example, in the expression $(\lambda x.\lambda y.$**if** `not` x **then** `succ` y **else** `pred` $y)$ `1 True`, where `not` has type `Bool` $\rightarrow$ `Bool` and `succ` and `pred` have type `Int` $\rightarrow$ `Int`, SCV will detect the failure at `not` x only and will not report the possible failures in the branches. In contrast, our approach can detect all inconsistencies due to the error tolerance of variational typing.

Overall, our work and SCV seem to be complementary.

9 CONCLUSION

We have presented a methodology for detecting, understanding, and fixing inconsistencies in gradually typed programs. To support this, we designed exploratory typing to efficiently explore all possible ways of adding, removing, and replacing type annotations for a given program. We have implemented exploratory typing and our algorithms for supporting repair steps S1-S3 in PyHound. Our evaluation on 282 programs shows that PyHound effectively meets its goals, realizing repair steps S1-S3 in more than 93% of cases, significantly outperforming the widely used Python type annotation enforcement tool pytype. Moreover, PyHound is efficient, scaling linearly with program size and having less than eight times the overhead of Reticulated's type-checking, even for programs with 15,000 LOC. In the future, we plan to explore the applicability of exploratory typing to other gradually typed languages.

REFERENCES

2020. Pytype. https://github.com/google/pytype Last accessed on April 27th, 2020.

Martín Abadi, Luca Cardelli, Benjamin Pierce, and Gordon Plotkin. 1991. Dynamic Typing in a Statically Typed Language. *ACM Trans. Program. Lang. Syst.* 13, 2 (April 1991), 237–268. https://doi.org/10.1145/103135.103138

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

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

Alex Aiken and Brian Murphy. 1991. Static Type Inference in a Dynamically Typed Language. In *Proceedings of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '91)*. ACM, New York, NY, USA, 279–290. https://doi.org/10.1145/99583.99621

Felipe Bañados Schwerter, Ronald Garcia, and Éric Tanter. 2014. A Theory of Gradual Effect Systems. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming (ICFP '14)*. ACM, New York, NY, USA, 283–295. https://doi.org/10.1145/2628136.2628149

Felipe Bañados Schwerter, Ronald Garcia, and Éric Tanter. 2016. Gradual type-and-effect systems. *Journal of Functional Programming* 26 (2016), e19. https://doi.org/10.1017/S0956796816000162

Johannes Bader, Jonathan Aldrich, and Éric Tanter. 2018. Gradual Program Verification. In *Verification, Model Checking, and Abstract Interpretation*, Isil Dillig and Jens Palsberg (Eds.). Springer International Publishing, Cham, 25–46.

Bernd Braßel. 2004. TypeHope: There is hope for your type errors. In *Int. Workshop on Implementation of Functional Languages*.

John Campora, Sheng Chen, Martin Erwig, and Eric Walkingshaw. 2018b. Migrating Gradual Types. In *Proceedings of the 45th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL '18)*. ACM, New York, NY, USA.

John Peter Campora, Sheng Chen, and Eric Walkingshaw. 2018a. Casts and Costs: Harmonizing Safety and Performance in Gradual Typing. *Proc. ACM Program. Lang.* 2, ICFP, Article 98 (July 2018), 30 pages. https://doi.org/10.1145/3236793

Robert Cartwright and Mike Fagan. 1991. Soft Typing. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation (PLDI '91)*. ACM, New York, NY, USA, 278–292. https://doi.org/10.1145/113445.113469

Giuseppe Castagna and Victor Lanvin. 2017. Gradual Typing with Union and Intersection Types. In *ACM SIGPLAN International Conference on Functional Programming (ICFP 2017)*. To appear.

Avik Chaudhuri, Panagiotis Vekris, Sam Goldman, Marshall Roch, and Gabriel Levi. 2017. Fast and Precise Type Checking for JavaScript. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 48 (Oct. 2017), 30 pages. https://doi.org/10.1145/3133872

S. Chen, M. Erwig, and E. Walkingshaw. 2012a. An Error-Tolerant Type System for Variational Lambda Calculus. In *ACM Int. Conf. on Functional Programming*. 29–40.

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

S. Chen, M. Erwig, and E. Walkingshaw. 2014a. Extending Type Inference to Variational Programs. *ACM Trans. on Programming Languages and Systems* 36, 1, Article 1 (2014), 54 pages.

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

Tim Disney and Cormac Flanagan. 2011. Gradual Information Flow Typing. In *International Workshop on Scripts to Programs*.

Oli Evans and Alan Shaw. 2018. asciify. https://github.com/olizilla/asciify

Asger Feldthaus and Anders Møller. 2014. Checking Correctness of TypeScript Interfaces for JavaScript Libraries. *SIGPLAN Not.* 49, 10 (Oct. 2014), 1–16. https://doi.org/10.1145/2714064.2660215

Luminous Fennell and Peter Thiemann. 2013. Gradual Security Typing with References. In *Proceedings of the 2013 IEEE 26th Computer Security Foundations Symposium (CSF '13)*. IEEE Computer Society, Washington, DC, USA, 224–239. https://doi.org/10.1109/CSF.2013.22

Ronald Garcia and Matteo Cimini. 2015. Principal Type Schemes for Gradual Programs. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '15)*. ACM, New York, NY, USA, 303–315. https://doi.org/10.1145/2676726.2676992

Ronald Garcia, Alison M. Clark, and Éric Tanter. 2016. Abstracting Gradual Typing. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '16)*. ACM, New York, NY, USA, 429–442. https://doi.org/10.1145/2837614.2837670

Ronald Garcia, Éric Tanter, Roger Wolff, and Jonathan Aldrich. 2014. Foundations of Typestate-Oriented Programming. *ACM Trans. Program. Lang. Syst.* 36, 4, Article 12 (Oct. 2014), 44 pages. https://doi.org/10.1145/2629609

Fritz Henglein. 1994. Dynamic Typing: Syntax and Proof Theory. In *Selected Papers of the Symposium on Fourth European Symposium on Programming (ESOP'92)*. Elsevier Science Publishers B. V., Amsterdam, The Netherlands, The Netherlands,

197–230. http://dl.acm.org/citation.cfm?id=197475.190867

David Herman, Aaron Tomb, and Cormac Flanagan. 2010. Space-efficient Gradual Typing. *Higher Order Symbol. Comput.* 23, 2 (June 2010), 167–189. https://doi.org/10.1007/s10990-011-9066-z

Atsushi Igarashi, Peter Thiemann, Vasco Vasconcelos, and Philip Wadler. 2017b. Gradual Session Types. In *ACM SIGPLAN International Conference on Functional Programming (ICFP 2017)*. To appear.

Yuu Igarashi, Taro Sekiyama, and Atsushi Igarashi. 2017a. On Polymorphic Gradual Typing. *Proc. ACM Program. Lang.* 1, ICFP, Article 40 (Aug. 2017), 29 pages. https://doi.org/10.1145/3110284

Lintaro Ina and Atsushi Igarashi. 2011. Gradual Typing for Generics. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '11)*. ACM, New York, NY, USA, 609–624. https://doi.org/10.1145/2048066.2048114

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

Andrew M. Kent, David Kempe, and Sam Tobin-Hochstadt. 2016. Occurrence Typing Modulo Theories. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '16)*. ACM, New York, NY, USA, 296–309. https://doi.org/10.1145/2908080.2908091

Kenneth Knowles and Cormac Flanagan. 2010. Hybrid Type Checking. *ACM Trans. Program. Lang. Syst.* 32, 2, Article 6 (Feb. 2010), 34 pages. https://doi.org/10.1145/1667048.1667051

Erik Krogh Kristensen and Anders Møller. 2017a. Inference and Evolution of TypeScript Declaration Files. In *Fundamental Approaches to Software Engineering*, Marieke Huisman and Julia Rubin (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 99–115.

Erik Krogh Kristensen and Anders Møller. 2017b. Type Test Scripts for TypeScript Testing. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 90 (Oct. 2017), 25 pages. https://doi.org/10.1145/3133914

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

Phúc C. Nguyen, Sam Tobin-Hochstadt, and David Van Horn. 2014. Soft Contract Verification. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming (ICFP '14)*. ACM, New York, NY, USA, 139–152. https://doi.org/10.1145/2628136.2628156

Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Mark Shields. 2007. Practical Type Inference for Arbitrary-rank Types. *J. Funct. Program.* 17, 1 (Jan. 2007), 1–82.

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

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

Jeremy Siek, Michael M. Vitousek, Matteo Cimini, Sam Tobin-Hochstadt, and Ronald Garcia. 2015b. Monotonic References for Efficient Gradual Typing. https://doi.org/10.1007/978-3-662-46669-8_18

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

Jeremy G. Siek and Manish Vachharajani. 2008. Gradual Typing with Unification-based Inference. In *Proceedings of the 2008 Symposium on Dynamic Languages (DLS '08)*. ACM, New York, NY, USA, Article 7, 12 pages. https://doi.org/10.1145/1408681.1408688

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

Vincent St-Amour and Neil Toronto. 2013. Experience Report: Applying Random Testing to a Base Type Environment. In *ACM Int. Conf. on Functional Programming*. 351–356.

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

Satish Thatte. 1988. Type inference with partial types. In *Automata, Languages and Programming*, Timo Lepistö and Arto Salomaa (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 615–629.

Sam Tobin-Hochstadt and Matthias Felleisen. 2006. Interlanguage Migration: From Scripts to Programs. In *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications (OOPSLA '06)*. ACM, New York, NY, USA, 964–974. https://doi.org/10.1145/1176617.1176755

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

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

Matías Toro, Elizabeth Labrada, and Éric Tanter. 2019. Gradual Parametricity, Revisited. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '19)*. To appear.

Matías Toro and Éric Tanter. 2017. A Gradual Interpretation of Union Types. In *SAS*.

Preston Tunnell Wilson, Ben Greenman, Justin Pombrio, and Shriram Krishnamurthi. 2018. The Behavior of Gradual Types: A User Study. In *Proceedings of the 14th ACM SIGPLAN International Symposium on Dynamic Languages (DLS 2018)*. ACM, New York, NY, USA, 1–12. https://doi.org/10.1145/3276945.3276947

Michael M. Vitousek, Andrew M. Kent, Jeremy G. Siek, and Jim Baker. 2014. Design and Evaluation of Gradual Typing for Python. (2014), 45–56.

Michael M. Vitousek, Cameron Swords, and Jeremy G. Siek. 2017. Big Types in Little Runtime: Open-world Soundness and Collaborative Blame for Gradual Type Systems. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017)*. ACM, New York, NY, USA, 762–774. https://doi.org/10.1145/3009837.3009849

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

Jack Williams, J. Garrett Morris, Philip Wadler, and Jakub Zalewski. 2017. Mixed Messages: Measuring Conformance and Non-Interference in TypeScript. In *31st European Conference on Object-Oriented Programming (ECOOP 2017) (Leibniz International Proceedings in Informatics (LIPIcs))*, Peter Müller (Ed.), Vol. 74. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 28:1–28:29. https://doi.org/10.4230/LIPIcs.ECOOP.2017.28

Roger Wolff, Ronald Garcia, Éric Tanter, and Jonathan Aldrich. 2011. Gradual Typestate. In *Proceedings of the 25th European Conference on Object-oriented Programming (ECOOP'11)*. Springer-Verlag, Berlin, Heidelberg, 459–483. http://dl.acm.org/citation.cfm?id=2032497.2032529

Baijun Wu and Sheng Chen. 2017. How Type Errors Were Fixed and What Students Did? *Proc. ACM Program. Lang.* 1, OOPSLA, Article 105 (Oct. 2017), 27 pages. https://doi.org/10.1145/3133929

Ningning Xie, Xuan Bi, and Bruno C. d. S. Oliveira. 2018. Consistent Subtyping for All. In *Programming Languages and Systems*, Amal Ahmed (Ed.). Springer International Publishing, Cham, 3–30.