JFP **28**, e2, 48 pages, 2018.
doi:10.1017/S095679681700020X

*Systematic identification and communication of type errors**

SHENG CHEN
UL Lafayette
(*e-mail:* chen@louisiana.edu)

MARTIN ERWIG
Oregon State University
(*e-mail:* erwig@oregonstate.edu)

Abstract

When type inference fails, it is often difficult to pinpoint the cause of the type error among many potential candidates. Generating informative messages to remove the type error is another difficult task due to the limited availability of type information. Over the last three decades many approaches have been developed to help debug type errors. However, most of these methods suffer from one or more of the following problems: (1) Being incomplete, they miss the real cause. (2) They cover many potential causes without distinguishing them. (3) They provide little or no information for how to remove the type error. Any one of this problems can turn the type-error debugging process into a tedious and ineffective endeavor. To address this issue, we have developed a method named *counter-factual typing*, which (1) finds a comprehensive set of error causes in AST leaves, (2) computes an informative message on how to get rid of the type error for each error cause, and (3) ranks all messages and iteratively presents the message for the most likely error cause. The biggest technical challenge is the efficient generation of all error messages, which seems to be exponential in the size of the expression. We address this challenge by employing the idea of variational typing that systematically reuses computations for shared parts and generates all messages by typing the whole ill-typed expression only once. We have evaluated our approach over a large set of examples collected from previous publications in the literature. The evaluation result shows that our approach outperforms previous approaches and is computationally feasible.

1 Introduction

Static typing is one of the most widely used formal methods. A type system is an effective, but not too intrusive, method to catch a wide range of programming errors. Type inference increases the usability of type systems by allowing programmers to omit type annotations.

However, when type inference fails, it is often difficult to precisely locate the erroneous code and produce a suggestion to fix the error. This problem was

* This work is supported by the National Science Foundation under the grants IIS-1314384, CCF-1717300, and CCF-1750886. We thank the anonymous POPL and JFP reviewers, whose feedback has improved both the content and the presentation of the paper.

recognized by researchers (Johnson & Walz, 1986; Wand, 1986) soon after the invention of the algorithm $\mathcal{W}$ (Damas & Milner, 1982), the classic type inference algorithm underlying many modern type checkers for functional languages. Many researchers view fixing type errors in response to obscure and misleading error messages as one of the biggest obstacles to beginners to functional programming (Heeren *et al.*, 2003; Neubauer & Thiemann, 2003). Recent studies have confirmed this view (Chambers *et al.*, 2012; Tirronen *et al.*, 2015; Wu & Chen, 2017).

1.1 The difficulty of debugging type errors

In the last three decades, numerous approaches have been developed to assist the type error debugging process (Lee & Yi, 1998; Lee & Yi, 2000; Tip & Dinesh, 2001; Yang, 2001; Choppella, 2002; McAdam, 2002a; McAdam, 2002b; Haack & Wells, 2003; Heeren, 2005; Wazny, 2006; Lerner *et al.*, 2007; Schilling, 2012; Zhang & Myers, 2014; Loncaric *et al.*, 2016; Chen *et al.*, 2017; Wu *et al.*, 2017). Although considerable progress has been made, generating informative and helpful type error messages is still a challenge. In particular, there is still no single method that consistently produces satisfactory results. Most of the existing approaches perform poorly in certain cases.

As an example, consider the following Haskell function `palin`, which checks whether a list is a palindrome (Stuckey *et al.*, 2003). The first equation for `fold` contains a type error and should return `z` instead of `[z]`.

```
fold f z []     = [z]
fold f z (x:xs) = fold f (f z x) xs

flip f x y = f y x
rev = fold (flip (:)) []

palin xs = rev xs == xs
```

Existing tools have difficulties in finding this error. For example, the Glasgow Haskell Compiler (GHC) 7.8[1] produces the following error message.[2]

```
Occurs check: cannot construct the infinite type: t ~ [t]
Expected type: [[t]]
  Actual type: [t]
Relevant bindings include
  xs :: [t] (bound at Palin.hs:5:7)
  palin :: [t] -> Bool (bound at Palin.hs:5:1)
In the second argument of (==), namely ‘xs’
In the expression: rev xs == xs
```

GHC 8.0.2, the latest version of GHC at the time of writing, produces the same message. Hugs98,[3] another modern Haskell compiler, displays the following message:

[1] `www.haskell.org/ghc/`

[2] For presentation purposes, we have slightly edited the outputs of some tools by changing their indentation and line breaks.

[3] `http://www.haskell.org/hugs/`

```
*** Expression     : rev xs == xs
*** Term           : rev xs
*** Type           : [[a]]
*** Does not match : [a]
*** Because        : unification would give infinite type
```

We observe similar error messages from both compilers. While technically accurate, the error messages do not directly point to the source of the error, and they do not tell the user how the type error could be fixed either. The use of compiler jargon makes the error messages difficult to understand for many programmers. While giving reasons for the failure of unification might be useful for experienced programmers and type system experts, such error messages still require efforts to manually reconstruct some of the types and solve unification problems.

In general, compilers perform type inference by traversing the expression from left to right and generating and solving typing constraints at each node.[4] This approach suffers from three problems as far as error debugging is concerned. First, it tends to locate the type error to the right of the real cause in the AST (*right bias*). Second, it attributes each type error to a single subexpression (*single-error limitation*). In many cases, such reported locations are incorrect, since the program text does not contain enough information to confidently make the right decision about the correct error location. Third, it reports type errors in terms of unification failures and does not suggest changes to fix the type error (*lack of suggestions*).

In addition to the fixed order of traversing expressions, the principality of type inference also contributes to the right bias problem. In type inference, the types for all subexpressions should be as general as possible, captured in principal types. For example, the function `id` should have the type `a -> a`, rather than a more specific type, to make both `id 1` and `id True` well typed. In general, principality implies that, in $e_1\ e_2$, any subexpression in e_1 should not be blamed as an error cause as long as e_1 is well-typed, even though the real cause may be that some subexpression inside e_1 is too restrictive. This is exactly why both GHC and Hugs98 identify the error in the body of `palin` rather than in `fold`.

Many approaches have been proposed to address the right bias problem with the standard type inference model. They all share the idea of traversing the expression in a different order to eliminate the bias (Lee & Yi, 2000; Yang *et al*., 2000; McAdam, 2002b; Eo *et al*., 2004). However, while eliminating right bias, they introduce other biases. In fact, based on our analysis in the last paragraph, biases cannot be fully eliminated unless we forgo principality, the principle underlying type inference.

The right bias and single-error limitation problems with the standard type inference model have been tackled by a number of program slicing approaches. These try to identify a *set* of possible error locations rather than commit to a single location. The basic idea is to find *all* program positions that contribute to a type error and exclude those that do not. For example, the Skalpel[5] type error slicer for SML (Haack & Wells, 2003) produces the following result. (We have translated the

[4] Our discussion applies even when constraint generating and solving are separated into two phases.
[5] `www.macs.hw.ac.uk/ultra/skalpel/`

above program into ML for Skalpel to work.) All the parts that contribute to the type error are shaded.

```
fun fold f z  []      = [z] ;
  | fold f z  (x::xs) = fold f  (f (z,x)) xs ;
fun flip f (x,y) = f (y, x) ;
fun rev xs = fold ( (flip op ::)) [] xs ;
fun palin  xs = rev xs = xs  ;
```

Skalpel finds type error slices in two steps. First, it generates labeled constraints for programs such that constraint solving failure can be linked back to programs. Second, it finds a minimal unsolvable constraint set if the generated constraints could not be solved successfully. A significant difference of its constraint generation process compared to other inference algorithms is that to represent the result type of a subexpression, it creates a fresh variable and a constraint equating that fresh variable and the result type. This ensures that all subexpressions involved in the type error will be included in the slice. However, this strategy also causes the slice to include unnecessary subexpressions. For example, the variables defined for passing around an erroneous expression will be included in the slice. As we can see from the example, Skalpel includes most program parts in the slice.

Showing too many program locations involved in the type error diminishes the value of the slicing approach because of the cognitive burden put on the programmer to work through all marked code and to single out the proper error location. To address this problem, techniques have been developed that try to minimize the possible locations contributing to a type error. An example is the Chameleon Type Debugger,[6] which produces the following output:

```
fold f z []    = [z] ;
fold f z (x:xs)= fold f (f z x) xs ;
flip f x y = f y x ;
rev = fold (flip (:)) [] ;
palin xs = rev xs == xs ;
```

Chameleon is based on constraint handling rules and identifies a minimal set of unsatisfiable constraints, from which the corresponding places in the program contributing to the type error are derived.

Following this idea of showing fewer locations, a more aggressive strategy is taken by SHErrLoc,[7] which analyzes all the constraints and identifies the constraint that is most likely to cause the error based on a simple Bayesian model (Zhang & Myers, 2014; Zhang *et al.*, 2015). For the `palin` example, it outputs the following message:

[6] `ww2.cs.mu.oz.au/ sulzmann/chameleon/`. Since Chameleon doesn't offer a type diagnosis option anymore, the result is reproduced directly from Stuckey *et al.* (2003).

[7] `http://www.cs.cornell.edu/projects/SHErrLoc/`

```
Constraints in the source code that appear most likely to be wrong
(mouse over to highlight code):

(variable)t_aU9 == (variable)t_aUf [loc: program.hs:3,19-20]{z in [z]}
(variable)t_aTX == (variable)t_aU0 [loc: program.hs:5,16-17]{y}
(variable)t_aUi == (variable)t_aUs [loc: program.hs:4,30-31]{x in (f z x)}

4 errors found

A value with type a_aUJ is being used at type a_aUJ
fold f z []     = [z]
fold f z (x:xs)= fold f (f z x) xs
flip f x y = f y x
rev = fold (flip (:)) []
palin xs = rev xs == xs
Three other messages and their associated slices omitted
```

SHErrLoc's response consists of two parts. The first presents the constraints that are most likely to cause the type error. We have attached the highlighted code in italics in a pair of curly brackets to each corresponding constraint. Note that the first constraint mentions `z`, which is just part of the real error cause `[z]`, according to Stuckey *et al.* (2003). The second part presents the likely errors and related slices.

While SHErrLoc is a clear improvement over other slicing approaches in reporting fewer potential error locations, a programmer still has to work through several code parts to find the type error. In particular, figuring out which types should be used at specific locations can be quite time consuming.

Many techniques have been developed to address the lack-of-suggestions problem of the standard type inference model. They locate the most likely cause of the type error based on heuristics and provide change suggestions in some form. One example is the Helium compiler,[8] which was developed to support the teaching of typed functional programming languages. A declared focus of Helium is to generate good error messages (Heeren, 2005). In many cases, Helium produces high quality type error messages. For our example, it produces the following message:

```
(5,19): Type error in infix application
 expression       : rev xs == xs
 operator         : ==
   type           : a     -> a   -> Bool
   does not match : [[b]] -> [b] -> c
 because          : unification would give infinite type
```

Unfortunately, Helium does not perform much better than GHC or Hugs on this example and provides feedback only in terms of internal representations used by the compiler.

Helium locates type errors in several steps. First, it builds a type graph (McAdam, 1999; Erwig, 2006) to represent the constraints about typing relations among different parts of the program. When constraint solving fails, it will search through the type graph to discover a constraint whose removal will cause the constraint solving to

[8] `www.cs.uu.nl/wiki/bin/view/Helium/WebHome`

```
File "Palin.ml", line 8, characters 21-27:
This expression has type 'a list list but is
here used with type 'a list
Relevant code: rev xs
------------------------------------------------
File "Palin.ml", line 8, characters 15-17:
Try replacing
    xs == (rev xs)
with
    ( == ) (xs, (rev xs))
of type
    'b list * 'b list list -> bool
within context
    let palin xs = ( == ) (xs, (rev xs)) ;;
```

Fig. 1. The error message of Seminal for `palin`.

succeed. Helium finds the most suspicious constraint based on some heuristics (Hage & Heeren, 2007).

Seminal,[9] a tool for type checking OCaml programs, also produces change suggestions (Lerner *et al*., 2006, 2007). Seminal blames the type error on the function `palin` and produces a corrective change shown in Figure 1.

Unfortunately, the suggested error location is not correct according to Stuckey *et al.* (2003). Although the suggested change will eliminate the type error, it changes the wrong code: The suggested change of partially applying `==` to the pair of differently typed lists turns `palin`'s type into `'a list -> 'a list * 'a list list -> Bool`.

Seminal locates type errors using binary search. In the example, it first type checks the first half of the program, which does not contain a type error when considered alone. It then concludes that the type error is in the second half and directs the search process to that part. Following this search strategy, the type error will be found in the last line because if we remove that line, the remaining program is well typed. This shows that Seminal does not work well when the cause of a type error occurs in a subexpression that is itself well typed but causes type errors in other subexpressions.

Offering change suggestions is a double-edged sword: While it can be very helpful in simplifying the task of fixing type errors, it can also be sometimes very misleading and frustrating when the suggested change does not work. In the shown example, both change-suggesting tools fail to correctly locate the error, and consequently their change suggestions also fail.

Overall, none of the existing approaches adequately addresses all the problems with the standard type inference model. While error slicing approaches tackle the right bias and single-error limitation problems, they do not address the lack-of-suggestions problem. Moreover, they are too general, and their improved variants can again miss the real error cause. While change-suggestion tools obviously address the latter problem, they generally do not address right bias and single-error limitation.

[9] `cs.brown.edu/~blerner/papers/seminal_prototype.html`

In some cases, both Helium and Seminal can report errors at more than one location. However, in general they report only few locations, and the identified sets of causes are incomplete.[10]

1.2 Systematic identification and communication of type errors

The task of debugging type errors seems to be an inherently ambiguous undertaking, because in some situations there is just not enough information present in the program to generate a correct change suggestion. Consider, for example, the following expression.

```
not 1
```

The error in this expression is either `not` or `1`,[11] but without any additional knowledge about the purpose of the expression, there is no way to decide whether to replace the function or the argument. This is why it is generally impossible to isolate one point in the program as the source of a type error. This observation provides strong support for slicing approaches that try to provide an unbiased account of error situations. On the other hand, in many cases some locations are more likely than others, and specifically in larger programs, information about the context of an erroneous expression can go a long way in isolating a single location for a type error.

Thus, a reasonable compromise between slicing and single-error-reporting approaches could be a method to principally compute *all* possible type error locations (together with possible change suggestions) and present them ranked and in small portions to the programmer. This idea addresses all the three problems mentioned above and avoids the shortcoming of error slicing approaches of not being specific. The biggest challenge in realizing this idea is its high complexity. To find the change suggestion for each location, or any combination of the locations, we potentially have to repeatedly modify the program and run the type inference process. The complexity of this naive approach is exponential in the size of the input program.

To address this challenge, we propose a method that identifies type errors by systematically varying the program being checked. We call this method *counterfactual* (CF) *change inference*, whose core is a technique to answer the question "What type should a particular subexpression have to remove type errors in a program." We have implemented and evaluated a prototype for a type checker that is based on this technique.

For each identified type error, the following information will be computed:

- The location of the erroneous expression that needs to be changed. When fixing the type error needs to change multiple locations, all of them will be identified.

[10] The approach we will present is provably complete, yet our algorithm sacrifices it for performance in large programs. In practice, this does not have an adverse effect on the precision of our algorithm. We discuss this in detail in Sections 5 and 7.

[11] It could also be the case that the whole expression is incorrect and should be replaced by something else, but we ignore this case for now.

Rank	Loc	Code	Change code of type/ Change expression	To new type/ To new expression	Result type
1	(1,19)	`(:)`	`a -> [a] -> [a]` `[z]`[12]	`a -> [b] -> a` `z`	`[a] -> Bool`
2	(5,22)	`xs`	`[a]`	`[[a]]`	`[a] -> Bool`
3	(5,12)	`rev`	`[a] -> [[a]]`	`[a] -> [a]`	`[a] -> Bool`
4	(5,19)	`(==)`	`[a] -> [a] -> Bool`	`[[a]] -> [a] -> b`	`[a] -> b`
5	(4,7)	`fold`	`t1`	`t2`	`[a] -> Bool`

Fig. 2. Ranked list of the first five single-location type and expression change suggestions inferred for the `palin` example. The types `t1` and `t2` in the table are `(a -> b -> a) -> a -> [b] -> [a]` and `([a] -> a -> [a]) -> [b] -> [a] -> [a]`, respectively. The first two lines denote a single fix, where the expression change in the second line is deduced from the type change from the first line. We discuss the deduction process in Section 6.

- The type of the identified erroneous subexpression under normal type inference.
- The expected type each identified erroneous expression ought to have to remove the type error.
- The result type of the expression if all the identified subexpressions are changed to the expected types.

To keep the complexity manageable we only produce *type changes* at AST leaves. Here "type" indicates that the change information is about the types of program parts, for example the inferred type and the expected type of the erroneous leaf. An example type change for `palin` is

Change `(:)` of type `a -> [a] -> [a]` to something of type `a -> [b] -> a`

However, errors that are best fixed by *expression changes* are quite common. Examples are the swapping of function arguments or the addition of missing function arguments. In such cases, where the underlying change to the program is clear, we can directly report an expression change instead of a type change. An example for the `palin` program is

Change `[z]` to `z`

The identification of such expression changes is not precluded by our approach that focuses on identifying corrective type changes. In fact, expression changes can often be deduced from type changes. We discuss an approach of deducing expression changes from type changes in Section 6. Overall, our approach produces two kinds of change suggestions, type changes for AST leaves (potentially multiple leaves for fixing a single error) and expression changes for non-leaves. Since we never generate type changes for non-leaves and expression changes for leaves, we will from now on simply use the terms type changes and expression changes, without the qualifications "for AST leaves" and "for non-leaves," respectively.

Returning to the `palin` example, Figure 2 shows the ranked list of the first five[12] (single-location) type changes that are computed by our prototype and that can fix

[12] We omitted the remaining eight for clarity.

the type error. The correct change ranks first in our method. Note that this is not a representation intended to be given to end users. We rather envision an integration into a user interface in which locations are underlined and hovering over those locations with the mouse will pop up windows with individual change suggestions. In this paper, we focus on the technical foundation to compute the information required for implementing such a user interface.

Each suggestion is essentially represented by the expression that requires a change together with the inferred actual and expected type of that expression. Since we are only considering type changes at leaves, this expression will always be a constant or variable in case of a type change, but it can be a more complicated expression in case of a deduced expression change. We also show the position of the code in the program[13] and the result types of the program if the corresponding change is adopted. This information is meant as an additional guide for programmers to select among suggestions.

The list of shown suggestions is produced in several steps. First, we generate (lazily) all possible type changes, that is, even those that involve several locations. Note that sometimes the suggested types are unexpected. For example, the suggested type for `fold` is `([a] -> a -> [a]) -> [b] -> [a] -> [a]` although `(a -> b -> a) -> a -> [b] -> a` would be preferable. This phenomenon can be generally attributed to the context of the expression. On the one hand, the given context can be too restrictive and coerce the inferred type to be more specific than it has to be, just as in this example the first argument `flip (:)` forces `fold` to have `[a] -> a -> [a]` as the type of its first argument. On the other hand, the context could also be too unrestrictive. There is no information about how `fold` is related to `[]` in the fourth line of the program. Thus, the type of the second argument of `fold` is inferred as `[b]`. This imprecision cannot be remedied by exploiting type information of the program. We show how to generate such type changes with a small example in Section 3 and present general rules in Section 4.2 and an inference algorithm in Section 5.

Second, we filter out those type changes that involve only one location. We present those first to the programmer, since these are generally easier to understand and to adopt than multi-location change suggestions. Should the programmer reject all these single-location suggestions, two-location suggestions will be presented next, and so on.

Third, in addition to type-change suggestions, we also try to infer some expression changes from type changes. In general, only the programmer who wrote the program knows how to translate required type changes into expression changes. However, there are a number of common programming mistakes, such as swapping or forgetting arguments, that are indicated by type-change suggestions. Similar to Seminal, our prototype identifies these kind of changes that are mechanical and do not require a deep understanding of the program semantics. In our example, we infer the replacement of `[z]` by `z`, because the expected type requires that the return type be the same as the first argument type. We thus suggest to use the first argument,

[13] We have added the line and column numbers by hand since our prototype currently works on abstract syntax and does not have access to the information from the parser.

that is `z`, to replace the application `(:) z []`. We present the deduction process in Section 6.

Note however, that we do not infer a similar change for the fifth type change because `fold` is partially applied in the definition, and we have no access to the third argument of `fold`. Had the `rev` function been implemented using an eta-expanded list argument, say `xs`, we would have also inferred the suggestion to change `fold (flip (:)) [] xs` to `xs`.

Note also that we do not supplement type changes at leaves with expression changes. For example, in the second suggestion, we do *not* suggest to replace `xs` by `[xs]`. There are two reasons for this. On the one hand, we believe that, given the very specific term to change, the inferred type, and the expected type, the corresponding required expression change is often easy to deduce for a programmer. On the other hand, suggesting specific expression changes requires knowledge about program semantics that is in many cases not readily available in the program. Thus, such suggestions can often be misleading.

Finally, all the type-change suggestions are ranked according to a few simple, but effective complexity heuristics. We present such heuristics in Section 5.

1.3 Structure of the paper

At the core of the proposed method of CF typing is a type system for inferring a set of type-change suggestions. This type system is described in detail in Section 4. We show that the type system generates a complete and correct set of type change suggestions.

The type system is based on a systematic variation of the types of AST leaves in a program. Therefore, some background information on how to represent variation in expressions and types, how to make use of it for the purpose of type inference, and what technical challenges this poses, is provided in Section 2. Equipped with the necessary technical background, the rationale behind type-change inference can then be explained on a high level in Section 3.

The algorithmic aspects of type-change inference and some measures for controlling runtime complexity are discussed in Section 5. We also briefly describe the set of heuristics that we use for ranking change suggestions. The method of deducing expression changes from type changes is discussed in Section 6. We have evaluated our prototype implementation by comparing it with three closely related tools and found that CF typing can generate correct change suggestions more often than the other approaches. The evaluation is described in Section 7. Related work is discussed in Section 8, and conclusions presented in Section 9 complete this paper.

1.4 Additions in the journal paper

This paper is based on our prior work (Chen & Erwig, 2014a) but contains the following additions and updates:

- We have expanded the evaluation by adding a comparison of CF typing and Seminal on debugging programs with multiple type errors in Section 7.2.

We have also extended the performance evaluation by considering programs from Hage (2013) in Section 7.3.
- We have corrected an error in the rule for typing let-expressions in Section 4.2.
- We have added a high-level overview of CF typing in Section 6.
- We have added complete proofs for all theorems except simple ones.
- We have expanded our analysis of existing approaches in Section 1.1.
- We have extended the discussion of variational typing to make this paper more self-contained.
- We have included a discussion of recent publications in the literature.

2 Variational typing

The idea of type-change inference is to explicitly represent and reason about discrepancies between inferred and expected types. This idea can be exploited in different ways, and the CF typing method presented in this paper is just one incarnation of this more general strategy. In this section, we will first introduce the idea of variational types and variational type inference. We will then develop some technical machinery that is needed for the formalization of CF typing.

First, the goal of CF typing is to generate suggestions for how to change types and expressions in a program to fix a type error. Both kinds of changes will be represented using the generic choice representation of the choice calculus that was introduced in Erwing & Walkingshaw (2011). The first application of this representation in the context of type checking was to extend type inference to program families (that is, a set of related programs) (Chen *et al.*, 2014). We will introduce the concept of *choices*, *variational expressions* and *types*, and some related concepts in Section 2.1.

Second, when type checking a program family, it is important to obtain the types of some programs (family members) even if the typing of other programs fails. This leads to the notion of *partial types*, *typing patterns*, and an associate method for *partial unification*. The application rule will be further generalized to accommodate partial types. We will explain these concepts in Section 2.2.

2.1 Variational expressions and variational types

The Choice Calculus (Erwig & Walkingshaw, 2011) provides a disciplined way of representing variation in software. The most important concept is the *named choice*, which can be used to represent variation points in both expressions and types. For example, the variational expression $e = \texttt{not}\ A\langle \texttt{1}, \texttt{True}\rangle$ contains the named choice $A\langle \texttt{1}, \texttt{True}\rangle$ that represents a choice between the two constants `1` and `True` as the argument to `not` .

The process of eliminating a variation point is called *selection* (Chen *et al.*, 2014). Selection takes a selector of the form $D.i$, where D is the name of a choice, traverses the expression and replaces each D choice with its corresponding ith alternative. We write $\lfloor e \rfloor_{D.i}$ for selecting e with $D.i$. In this paper, we are only concerned with binary choices, that is, i will be either 1 or 2. For example, selecting $A.1$ from e yields the plain expression `not 1`. Plain expressions are obtained after all

choices are eliminated from a variational expression. Note that variation points with different names vary independently of one another, and only those with the same name are synchronized during selection. For example, the variational expression $A\langle \texttt{odd}, \texttt{not}\rangle\ A\langle 1, \texttt{True}\rangle$ represents only the two expressions `odd 1` and `not True`, while $A\langle \texttt{odd}, \texttt{not}\rangle\ B\langle 1, \texttt{True}\rangle$ represents the four expressions `odd 1`, `odd True`, `not 1`, and `not True`.

Through the use of independent choices, variational programs can very quickly encode a huge number of programs that differ only slightly. Ensuring the type correctness of all selectable plain programs is challenging because the brute-force approach of generating and checking each variant individually is generally infeasible. Variational type inference (Chen *et al.*, 2014) solves this problem by introducing variational types and a method for typing variational programs in one run. The result of type inference applied to a variational program is a variational type, from which the types for individual program variants can be obtained with the same selection as the programs.

The syntax of variational types is shown below where α ranges over type variables, and γ ranges over type constants. The type $\bot$ denotes the occurrences of type errors and will be discussed in Section 2.2.

$$\phi \quad ::= \quad \gamma \mid \alpha \mid \phi \to \phi \mid D\langle \phi, \phi \rangle \mid \bot$$

For example, the variational expression $e = A\langle \texttt{odd}, \texttt{not}\rangle$ has the variational type $\phi = A\langle \texttt{Int} \to \texttt{Bool}, \texttt{Bool} \to \texttt{Bool}\rangle$. The most important property of variational typing is that the type for each plain expression selected from a variational expression can be obtained through the same selections from the corresponding variational type. For example, by selecting $A.2$ from e we obtain the expression `not`, which has the type $\texttt{Bool} \to \texttt{Bool}$ that is obtained by selecting $A.2$ from ϕ.

An important technical part of variational typing is the fact that the equivalence of choices is not merely syntactical, but governed by a set of equivalence rules, originally described in Erwing & Walkingshaw (2011). For example, $A\langle \texttt{Int}, \texttt{Int}\rangle$ is equivalent to `Int`, written as $A\langle \texttt{Int}, \texttt{Int}\rangle \equiv \texttt{Int}$, since either decision in A yields `Int`. The full set of rules defining the equivalence relation is given in Figure 3. The first three rules state that the type equivalence relation is reflexive, symmetric, and transitive. The rule E4 states that a choice is equivalent to either of its alternative if both alternatives are the same. Here the metavariable D ranges over dimension names. The rule E5 states that the type equivalence relation is a congruence, where $\phi[\ ]$ represents a type context into which we can plug a type. The rule E6 states that removing dead alternatives preserves the equivalence relation. The operation $\lfloor \phi_1 \rfloor_{D.1}$ in the first alternative of D removes all variants reachable by $D.2$. These variants are dead due to choice synchronization. Finally, rule E7 states that the function constructor and the choice constructor may be commuted while preserving the equivalence relation. Note that all occurrences of D must be substituted with a same choice name.

As a slightly more sophisticated example, we can show that choices can sometimes commute past each other. For example, $A\langle B\langle \phi_1, \phi_2\rangle, \phi_2\rangle \equiv B\langle A\langle \phi_1, \phi_2\rangle, \phi_2\rangle$ can be

$$\text{E1}\quad \phi \equiv \phi \qquad \text{E2}\quad \frac{\phi \equiv \phi_1}{\phi_1 \equiv \phi} \qquad \text{E3}\quad \frac{\phi \equiv \phi_1 \qquad \phi_1 \equiv \phi_2}{\phi \equiv \phi_2} \qquad \text{E4}\quad D\langle \phi, \phi \rangle \equiv \phi \qquad \text{E5}\quad \frac{\phi_1 \equiv \phi_2}{\phi[\phi_1] \equiv \phi[\phi_2]}$$

$$\text{E6}\quad D\langle \phi_1, \phi_2 \rangle \equiv D\langle \lfloor \phi_1 \rfloor_{D.1}, \lfloor \phi_2 \rfloor_{D.2} \rangle \qquad \text{E7}\quad D\langle \phi_1, \phi_2 \rangle \to D\langle \phi_3, \phi_4 \rangle \equiv D\langle \phi_1 \to \phi_3, \phi_2 \to \phi_4 \rangle$$

Fig. 3. Variational type equivalence.

shown as follows:

$$\begin{aligned}
& A\langle B\langle \phi_1, \phi_2 \rangle, \phi_2 \rangle \\
\equiv\ & B\langle A\langle B\langle \phi_1, \phi_2 \rangle, \phi_2 \rangle, A\langle B\langle \phi_1, \phi_2 \rangle, \phi_2 \rangle \rangle && \text{by E4 from right to left} \\
\equiv\ & B\langle \lfloor A\langle B\langle \phi_1, \phi_2 \rangle, \phi_2 \rangle \rfloor_{B.1}, \lfloor A\langle B\langle \phi_1, \phi_2 \rangle, \phi_2 \rangle \rfloor_{B.2} \rangle && \text{by E6 from left to right} \\
=\ & B\langle A\langle \phi_1, \phi_2 \rangle, A\langle \phi_2, \phi_2 \rangle \rangle && \text{by the definition of selection} \\
\equiv\ & B\langle A\langle \phi_1, \phi_2 \rangle, \phi_2 \rangle && \text{by E4 from left to right and E5}
\end{aligned}$$

A shortcoming of the variational typing approach is that it can succeed only if all variants are well typed, that is, it is impossible to assign a type to the variational expression $A\langle \texttt{odd}, \texttt{not} \rangle\ \texttt{1}$, even though one of its variants is type correct. Error-tolerant variational typing addresses this issue.

2.2 Error-tolerant variational typing

The idea of error-tolerant typing (Chen *et al.*, 2012) is to assign the type $\bot$ to program variants that contain type errors. The explicit representation of type errors via $\bot$ as ordinary types supports the continuation of the typing process in the presence of type errors. Moreover, each variational program can be typed, and the resulting variational type contains $\bot$ for all variants that are type incorrect and a plain type for all type-correct variants. For example, $A\langle \texttt{odd}, \texttt{not} \rangle\ \texttt{1}$ has the type $A\langle \texttt{Bool}, \bot \rangle$, which encodes exactly the types that we obtain if we generate and type each expression separately.

The most challenging part of the error-tolerant type system is the handling of function applications because type errors can be introduced in different ways. For example, the function might not have an arrow type, or the type of the argument might not match the argument type of the function. Moreover, we have to consider the case of partial matching, that is, in the case of variational types, the argument type of the function and the type of the argument might be compatible for some variants only. Deciding in such a case that the whole application is of type $\bot$ would be too restrictive. This challenge is addressed by the following typing rule (Chen *et al.*, 2012).

$$\frac{\Gamma \vdash e_1 : \phi_1 \qquad \Gamma \vdash e_2 : \phi_2 \qquad vt_2' \to \phi' = \uparrow(\phi_1) \qquad \pi = \phi_2' \bowtie \phi_2 \qquad \phi = \pi \lhd \phi'}{\Gamma \vdash e_1\ e_2 : \phi}$$

The first two premises retrieve the types for the function and argument. Unlike in the traditional application rule, however, we do not require ϕ_1 to be a function type. Instead, the third premise tries to lift ϕ_1 into an arrow type using a function $\uparrow$ that is defined at the top of Figure 4. Lifting specifically accounts for the case in

$$\text{Typing patterns } \pi ::= \bot \mid \top \mid D\langle \pi, \pi \rangle$$

$$\boxed{\uparrow : \phi \to \phi}$$

$$\begin{aligned}
\uparrow(\phi_1 \to \phi_2) &= \phi_1 \to \phi_2 \\
\uparrow(D\langle \phi_1 \to \phi_1', \phi_2 \to \phi_2' \rangle) &= D\langle \phi_1, \phi_2 \rangle \to D\langle \phi_1', \phi_2' \rangle \\
\uparrow(D\langle \phi_1, \phi_2 \rangle) &= \uparrow(D\langle \uparrow(\phi_1), \uparrow(\phi_2) \rangle) \\
\uparrow(\phi) &= \bot \to \bot \qquad \text{(otherwise)}
\end{aligned}$$

$$\boxed{\bowtie : \phi \times \phi \to \pi}$$

$$\begin{aligned}
\phi \bowtie \phi &= \top & D\langle \phi_1, \phi_2 \rangle \bowtie D\langle \phi_3, \phi_4 \rangle &= D\langle \phi_1 \bowtie \phi_3, \phi_2 \bowtie \phi_4 \rangle \\
\bot \bowtie \phi &= \bot & D\langle \phi_1, \phi_2 \rangle \bowtie \phi_3 &= D\langle \phi_1 \bowtie \phi_3, \phi_2 \bowtie \phi_3 \rangle \\
\phi \bowtie \bot &= \bot & \phi_1 \bowtie D\langle \phi_2, \phi_3 \rangle &= D\langle \phi_1 \bowtie \phi_2, \phi_1 \bowtie \phi_3 \rangle \\
\phi \bowtie \phi' &= \bot & \phi_1 \to \phi_2 \bowtie \phi_1' \to \phi_2' &= (\phi_1 \bowtie \phi_1') \otimes (\phi_2 \bowtie \phi_2')
\end{aligned}$$

$$\boxed{\lhd : \pi \times \phi \to \phi} \qquad \boxed{\otimes : \pi \times \pi \to \pi}$$

$$\begin{aligned}
\bot \lhd \phi &= \bot & \top \otimes \pi &= \pi \\
\top \lhd \phi &= \phi & \bot \otimes \pi &= \bot \\
D\langle \pi_1, \pi_2 \rangle \lhd \phi &= D\langle \pi_1 \lhd \phi, \pi_2 \lhd \phi \rangle & D\langle \pi_1, \pi_2 \rangle \otimes \pi_3 &= D\langle \pi_1 \otimes \pi_3, \pi_2 \otimes \pi_3 \rangle
\end{aligned}$$

Fig. 4. Operations for typing applications.

which ϕ_1 is a choice between arrow types, as in $\uparrow(A\langle \texttt{Bool} \to \texttt{Bool}, \texttt{Int} \to \texttt{Bool} \rangle) = A\langle \texttt{Bool}, \texttt{Int} \rangle \to A\langle \texttt{Bool}, \texttt{Bool} \rangle$, and it also deals with, and introduces if necessary, error types. For example, $\uparrow(A\langle \texttt{Int} \to \texttt{Bool}, \texttt{Int} \rangle)$ can succeed only by introducing an error type and thus yields $A\langle \texttt{Int}, \bot \rangle \to A\langle \texttt{Bool}, \bot \rangle$.

The fourth premise computes a typing pattern π that records to what degree (that is, in which variants) the type of e_2 matches the argument type of the (partial) arrow type obtained for e_1. As can be seen in Figure 4, a typing pattern is a (possibly deeply nested) choice of the two values $\bot$ (type error) and typing success ($\top$). The computation of a typing pattern proceeds by induction over the type structure of its arguments. Note that the definition given in Figure 4 contains overlapping patterns and assumes that more specific cases are applied before more general ones. When two rules are equally applicable, the computed result is equivalent modulo the $\equiv$ relation (Chen *et al.*, 2012).

For two plain types, matching reduces to checking equality. For example, $\texttt{Int} \bowtie \texttt{Int} = \top$ and $\texttt{Int} \bowtie \texttt{Bool} = \bot$. On the other hand, matching a plain type with a variational type results in a choice pattern. For example, $\texttt{Int} \bowtie D\langle \texttt{Int}, \texttt{Bool} \rangle = D\langle \top, \bot \rangle$.

Note that for two arrow types to be matched successfully, both their corresponding argument types and return types have to be matched successfully. There is no partial matching for function types. We define the operation $\otimes$ to achieve this, which is also presented in Figure 4. We can essentially view it as the logical "and" operation if we treat $\top$ as true and $\bot$ as false. For example, when computing $\texttt{Int} \to A\langle \texttt{Bool}, \texttt{Int} \rangle \bowtie B\langle \texttt{Int}, \bot \rangle \to \texttt{Bool}$, we first obtain $B\langle \top, \bot \rangle$ and $A\langle \top, \bot \rangle$ for

matching the argument types and return types, respectively. Next, we use $\otimes$ to derive the final result as $B\langle A\langle \top, \bot\rangle, \bot\rangle$.

In the fifth and final premise, the typing pattern is used to preserve the type errors that the fourth premise has potentially produced. This is done by "masking" the return type with the typing pattern. The masking operation essentially replaces all the $\top$s in the typing pattern with the variants in the return type and leaves all $\bot$s unchanged, denoting the occurrences of type errors.

To see the application rule in action, consider the expression $A\langle \texttt{not}, \texttt{odd}\rangle\ B\langle \texttt{1}, \texttt{True}\rangle$. The first two premises produce the following typing judgments.

$$\begin{aligned} A\langle \texttt{not}, \texttt{odd}\rangle &: A\langle \texttt{Bool} \to \texttt{Bool}, \texttt{Int} \to \texttt{Bool}\rangle \\ B\langle \texttt{1}, \texttt{True}\rangle &: B\langle \texttt{Int}, \texttt{Bool}\rangle \end{aligned}$$

Lifting transforms the type of the function into $A\langle \texttt{Bool}, \texttt{Int}\rangle \to A\langle \texttt{Bool}, \texttt{Bool}\rangle$, which is equivalent to $A\langle \texttt{Bool}, \texttt{Int}\rangle \to \texttt{Bool}$. The computation of $A\langle \texttt{Bool}, \texttt{Int}\rangle \bowtie B\langle \texttt{Int}, \texttt{Bool}\rangle$ yields $\pi = A\langle B\langle \bot, \top\rangle, B\langle \top, \bot\rangle\rangle$, and masking the return type of the function, `Bool`, with π yields $A\langle B\langle \bot, \texttt{Bool}\rangle, B\langle \texttt{Bool}, \bot\rangle\rangle$.

3 Counter-factual typing

The main idea behind CF typing is to systematically vary parts of the ill-typed program to find changes that can eliminate the corresponding type error(s) from the program. It is infeasible to apply this strategy directly on the expression level, since one could consider infinitely many changes in general. Therefore, we perform the variation on the type level. Basically, we ask for each AST leaf e the counter-factual question: *What type should e have to make the program well typed?*

The CF reasoning is built into the type checking process in the following way. To determine the type of an expression e, we first infer e's type, say ϕ. But then, instead of fixing this type, we leave the decision open and assume e to have the type $D\langle \phi, \alpha\rangle$, where D is a fresh choice name and α is a fresh type variable. By leaving the type of e open to revision we account for the fact that e may, in fact, be the source of a type error. By choosing a fresh type variable for e's alternative type, we enable type information to flow from the context of e to forge an alternative type ϕ' that fits into the context in case ϕ does not. If ϕ does fit the context, it is unifiable with ϕ', and the choice could in principle be removed. However, this is not really necessary (and we, in fact, don't do this) since in case of a type-correct program, we can find the type at the end of the typing process by simply selecting the first option from all generated choices.

Let us illustrate this idea with a simple example. Consider the expression $e = \texttt{not 1}$. If we vary the types of both `not` and `1`, we obtain the following typing judgments:

$$\begin{aligned} \texttt{not} &: A\langle \texttt{Bool} \to \texttt{Bool}, \alpha_1\rangle \\ \texttt{1} &: B\langle \texttt{Int}, \alpha_2\rangle \end{aligned}$$

where α_1 and α_2 represent the expected types of `not` and `1` according to their respective contexts. To find the types α_1 and α_2, we have to solve the following

unification problem:

$$A\langle \texttt{Bool} \to \texttt{Bool}, \alpha_1 \rangle \equiv^? B\langle \texttt{Int}, \alpha_2 \rangle \to \alpha_3$$

where α_3 denotes the result type of the application and $\equiv^?$ denotes that the unification problem is solved modulo the type equivalence relation mentioned in Section 2.1 rather than the usual syntactical identity.

Another subtlety of the unification problem is that two types may not be unifiable. In that case a solution to the unification problem consists of a so-called *partial unifier*, which is both most general and introduces as few errors as possible. The unification algorithm developed in Chen *et al.* (2012) achieves both these goals.

For the above unification problem, the following unifier is computed. The generality introduced by α_6 and α_7 ensures that only the second alternatives of choices A and B are constrained (Chen *et al.*, 2014).

$$\begin{aligned}&\{\alpha_1 \mapsto A\langle \alpha_6, B\langle \texttt{Int}, \alpha_4 \rangle \to \alpha_5 \rangle,\\ &\ \alpha_2 \mapsto B\langle \alpha_7, A\langle \texttt{Bool}, \alpha_4 \rangle \rangle,\\ &\ \alpha_3 \mapsto A\langle \texttt{Bool}, \alpha_5 \rangle\}\end{aligned}$$

Additionally, the unification algorithm returns a typing pattern that characterizes all the viable variants and helps to compute the result type of the varied expression. In this case, we obtain $A\langle B\langle \bot, \top \rangle, \top \rangle$. Based on the unifier and the typing pattern, we can compute that the result type of the varied expression of `not 1` is $\phi = A\langle B\langle \bot, \texttt{Bool} \rangle, \alpha_5 \rangle$. From the result type and the unifier, we can draw the following conclusions:

- If we do not change e, that is, if we select $A.1$ and $B.1$ from the variational type, the result type is $\bot$ (the variant corresponds to $A.1$ and $B.1$ in the result type), which reflects the fact that the original expression is ill typed.
- If we change `not` to some other expression f, that is, if we select variant $A.2$ and $B.1$ from the variational type, the result type will be α_5. Moreover, the type of f is obtained by selecting $A.2$ and $B.1$ from the type that α_1 is mapped to, which yields $\texttt{Int} \to \alpha_5$. In other words, by changing `not` to an expression of type $\texttt{Int} \to \alpha_5$, `not 1` becomes well typed. In the larger context, α_5 may be further constrained to have some other type.
- If we change `1` to some expression g, that is, if we select $A.1$ and $B.2$ from the variational type, then the result type becomes `Bool`.
- If we vary both `not` to f and `1` to g, which means to select $A.2$ and $B.2$ from the variational type, the result type is α_5. Moreover, from the unifier we know that f and g should have the types $\alpha_4 \to \alpha_5$ and α_4, respectively.

This gives us all type changes for the expression `not 1`. The combination of creating variations at the type level and variational typing provides an efficient way of finding all possible type changes.

Term variables	x, y, z	Value constants	c
Type variables	α, β	Type constants	γ

Expressions	e, f	$::=$	$c \mid x \mid \lambda x.e \mid e\ e \mid \texttt{let}\ x = e\ \texttt{in}\ e \mid$ $\texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e$

Monotypes	τ	$::=$	$\gamma \mid \alpha \mid \tau \to \tau$
Variational types	ϕ	$::=$	$\tau \mid \bot \mid D\langle \phi, \phi \rangle \mid \phi \to \phi$
Type schemas	σ	$::=$	$\phi \mid \forall \overline{\alpha}.\phi$
Selectors	s	$::=$	$D.i$

Type environments	Γ	$::=$	$\varnothing \mid \Gamma, x \mapsto \sigma$
Substitutions	η, θ	$::=$	$\varnothing \mid \eta, \alpha \mapsto \phi$
Choice environments	Δ	$::=$	$\varnothing \mid \Delta, (l, D\langle \phi, \phi \rangle)$

Fig. 5. Syntax of expressions, types, and environments.

4 Type-change inference

This section presents the type system that generates a complete set of corrective type changes. After defining the syntax for expressions and types in Section 4.1, we present the typing rules for type-change inference in Section 4.2. In Section 4.3, we investigate some important properties of the type-change inference system.

4.1 Syntax

We consider a type checker for lambda calculus with let-polymorphism. Figure 5 shows the syntax for the expressions, types, and meta environments for the type system. We employ the abbreviating notation $\overline{\alpha}$ for a sequence of type variables $\alpha_1, \ldots, \alpha_n$. Both the definitions of expressions and types are conventional, except for variational types, which introduce choice types and the error type.

We use l to denote program locations, in particular, leaves in ASTs. We assume that there is a function $\ell_e(f)$ that returns l for f in e. For presentation purposes, we assume that f uniquely determines a location. We may omit the subscript e when the context is clear. The exact definition of $\ell(\cdot)$ does not matter.

As usual, Γ binds type variables to type schemas for storing typing assumptions. We use η to denote type substitutions that map type variables to variational types. The metavariable θ ranges over type substitutions that are unifiers for unifiable types or partial unifiers for non-unifiable types. Finally, we use the choice environment Δ to associate choice types that were generated during the typing process with the corresponding location in the program. Operations on types can be lifted to Δ by applying them to the types in Δ. We stipulate the conventional definition of FV that computes the free type variables in types, type schemas, and type environments. We write $\eta_{/S}$ for $\{\alpha \mapsto \phi \in \eta \mid \alpha \notin S\}$.

The application of a type substitution to a type schema is written as $\eta(\sigma)$ and replaces free type variables in σ by the corresponding images in η. The definition is as follows.

$$\boxed{\Gamma \vdash e : \phi | \Delta}$$

$$\textsc{Con}\ \frac{c \text{ is of type } \gamma \qquad D \text{ fresh}}{\Gamma \vdash c : D\langle \gamma, \phi \rangle | \{(\ell(c), D\langle \gamma, \phi \rangle)\}}$$

$$\textsc{Var}\ \frac{\Gamma(x) = \forall \overline{\alpha}.\phi_1 \qquad D \text{ fresh} \qquad \phi = \{\overline{\alpha \mapsto \phi'}\}(\phi_1)}{\Gamma \vdash x : D\langle \phi, \phi_2 \rangle | \{(\ell(x), D\langle \phi, \phi_2 \rangle)\}}$$

$$\textsc{Unbound}\ \frac{x \notin dom(\Gamma) \qquad D \text{ fresh}}{\Gamma \vdash x : D\langle \bot, \phi \rangle | \{(\ell(x), D\langle \bot, \phi \rangle)\}}$$

$$\textsc{Abs}\ \frac{\Gamma, x \mapsto \phi \vdash e : \phi' | \Delta}{\Gamma \vdash \lambda x.e : \phi \to \phi' | \Delta}$$

$$\textsc{App}\ \frac{\Gamma \vdash e_1 : \phi_1 | \Delta_1 \qquad \Gamma \vdash e_2 : \phi_2 | \Delta_2 \qquad \phi_2' \to \phi' = \uparrow(\phi_1) \qquad \pi = \phi_2' \bowtie \phi_2 \qquad \phi = \pi \lhd \phi'}{\Gamma \vdash e_1\ e_2 : \phi | \Delta_1 \cup \Delta_2}$$

$$\textsc{Let}\ \frac{\Gamma, x \mapsto \phi_1 \vdash e_1 : \phi_1 | \Delta_1 \qquad \overline{\alpha} = FV(\phi_1) - FV(\Gamma) \qquad \Gamma, x \mapsto \forall \overline{\alpha}.\phi_1 \vdash e_2 : \phi_2 | \Delta_2 \qquad \pi = \lceil \phi_1 \rceil \qquad \phi = \pi \lhd \phi_2}{\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \phi | \Delta_1 \cup \Delta_2}$$

$$\textsc{If}\ \frac{(\Gamma \vdash e_i : \phi_i | \Delta_i)^{i:1..3} \qquad \pi_1 = \phi_1 \bowtie \texttt{Bool} \qquad \pi_2 = \phi_2 \bowtie \phi_3 \qquad \phi = \pi_1 \lhd (\pi_2 \lhd \phi_2)}{\Gamma \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : \phi | \Delta_1 \cup \Delta_2 \cup \Delta_3}$$

$$\lceil \bot \rceil = \bot \qquad \lceil D\langle \phi_1, \phi_2 \rangle \rceil = D\langle \lceil \phi_1 \rceil, \lceil \phi_2 \rceil \rangle$$
$$\lceil \tau \rceil = \top \qquad \lceil \phi_1 \to \phi_2 \rceil = \lceil \phi_1 \rceil \otimes \lceil \phi_2 \rceil$$

Fig. 6. Rules for type-change inference. The notations are defined in Figure 5.

$$\eta(\bot) = \bot \qquad \eta(\phi_1 \to \phi_2) = \eta(\phi_1) \to \eta(\phi_2)$$
$$\eta(\forall \overline{\alpha}.\phi) = \forall \overline{\alpha}.\eta_{/\overline{\alpha}}(\phi) \qquad \eta(\alpha) = \begin{cases} \alpha \text{ if } \alpha \notin dom(\eta) \\ \phi \text{ if } \alpha \mapsto \phi \in \eta \end{cases}$$
$$\eta(D\langle \phi_1, \phi_2 \rangle) = D\langle \eta(\phi_1), \eta(\phi_2) \rangle$$

Note that we do not consider variational polymorphic types. This is not a problem since we can always lift quantifiers out of choices. For instance, $D\langle \forall \alpha.\phi_1, \forall \beta.\phi_2 \rangle$ can be transformed to $\forall \alpha_1 \beta_1.D\langle \phi_1', \phi_2' \rangle$ with $\alpha_1 \notin FV(\phi_1)$ and $\beta_1 \notin FV(\phi_2)$, and $\phi_1' = \{\alpha \mapsto \alpha_1\}(\phi_1)$ and $\phi_2' = \{\beta \mapsto \beta_1\}(\phi_2)$.

4.2 Typing rules

Figure 6 presents the typing rules for inferring type changes. The typing judgment is of the form $\Gamma \vdash e : \phi | \Delta$ and produces as a result a variational type ϕ that represents all the typing "potential" for e plus a set of type changes Δ for the AST leaves in e that will lead to the types in ϕ.

During this phase, we compute type changes in the form of choices only for AST leaves, that is, constants and variables. This is reflected in the typing rules as we generate fresh choices in rules Con, Var, and Unbound. In each case, we place the actual type in the first alternative and an arbitrary type in the second alternative of the choice. When an unbound variable is accessed, it causes a type error. We thus put $\bot$ in the first alternative of the choice.

The rule Abs for abstractions is very similar to that in other type systems except that variables are bound to variational types. The rule App for typing applications is very similar to the application rule discussed in Section 2.2. The only difference is that the rule here keeps track of the information for Δ.

We need to pay special attention to let-expressions, where type errors can occur in different parts. For example, a type error can occur in either the body or the bound term or both. An interesting situation arises when a type error occurs in the bound term, but the binding is not used in the body. To illustrate, consider the following example:

```
v = let x = not 1 in 5.
```

To simplify the discussion, assume that none of `not`, `1`, and `5` is changed during the typing process. In other words, both alternatives for the choices created for `not`, `1`, and `5` are `Bool → Bool`, `Int`, and `Int`, respectively. Under this assumption, the expression should have the result type $\bot$, indicating an error when no changes are made to the expression. If we do not track the type error in `not 1`, then we may conclude that the whole expression is well typed because `5` is well typed.

To correctly track errors in the bound term, we define an auxiliary function $\lceil\cdot\rceil$ at the bottom of Figure 6. This function abstracts a variational type into a typing pattern by keeping all $\bot$s in the type and turning all other types into $\top$s. For example, $\lceil \texttt{Int} \rceil = \top$ and $\lceil D\langle \texttt{Int} \to \texttt{Bool}, \bot \rangle \rceil = D\langle \top, \bot \rangle$. With $\lceil\cdot\rceil$, we type let-expressions with the rule Let. The first three premises are standard. In the fourth premise, we compute where type errors have happened in the bound term with $\lceil\cdot\rceil$. After that, we mask the type of the body with the pattern from the previous premise, making sure that the errors from the bound term are transferred to the whole expression. We have defined the masking operation $\lhd$ in Figure 4.

Now reconsider typing the expression `v` with Let, assuming again that none of `not`, `1`, and `5` is changed. The type for `x` is $\bot$, yielding $\bot$ for π based on $\lceil\cdot\rceil$. Consequently, the result type (ϕ_2) is $\bot$. Overall, we observe that the error in the bound term is not lost in the result type.

The If rule employs the same machinery as the App rule for the potential introduction of type errors and partially correct types. In particular, the condition e_1 is not strictly required to have the type `Bool`. However, only the variants that are equivalent to `Bool` are type correct. Likewise, only the variants in which both branches are equivalent are type correct.

4.3 Properties

In this section, we investigate some important properties of the type-change inference system. We show that it is consistent in the sense that any type selected from the result variational type can be obtained by applying the changes as indicated by that selection. We also show that the type-change inference is complete in finding all corrective type changes for leaves. Based on this result, we show that the type-change inference system is a conservative extension of the Hindley–Milner type system (HM).

$$\boxed{\Delta \Downarrow \Delta'}$$

$$\varnothing \Downarrow \varnothing \qquad \frac{\Delta \Downarrow \Delta' \qquad \exists \tau \colon D\langle \overline{\phi} \rangle \equiv \tau}{\Delta, (l, D\langle \overline{\phi} \rangle) \Downarrow \Delta'} \qquad \frac{\Delta \Downarrow \Delta' \qquad \neg \exists \tau \colon D\langle \overline{\phi} \rangle \equiv \tau}{\Delta, (l, D\langle \overline{\phi} \rangle) \Downarrow \Delta', (l, D\langle \overline{\phi} \rangle)}$$

$$\boxed{\lfloor \_ \rfloor_{\_} : \phi \times s \to \phi}$$

$$\lfloor \tau \rfloor_s = \tau \qquad \lfloor \phi_1 \to \phi_2 \rfloor_s = \lfloor \phi_1 \rfloor_s \to \lfloor \phi_2 \rfloor_s$$

$$\lfloor \bot \rfloor_s = \bot \qquad \lfloor B\langle \overline{\phi} \rangle \rfloor_{A.i} = B\langle \overline{\lfloor \phi \rfloor_{A.i}} \rangle \quad \text{if } A \neq B$$

$$\lfloor B\langle \overline{\phi} \rangle \rfloor_{B.i} = \lfloor \phi_i \rfloor_{B.i}$$

Fig. 7. Simplifications and selection.

We start with the observation that type-change inference always succeeds in deriving a type for any given expression and type environment.

Lemma 1
Given e and Γ, there exist ϕ and Δ such that $\Gamma \vdash e : \phi | \Delta$.

The proof of this lemma is obvious because for any construct in the language, even for unbound variables, there is a corresponding typing rule in Figure 6 that applies and returns a type.

Next, we need to simplify Δ in the judgment $\Gamma \vdash e : \phi | \Delta$ to investigate the properties of the type system. Specifically, we define a simplification relation $\Downarrow$ in Figure 7 that eliminates idempotent choices from Δ. Note that the sole purpose of simplification is to eliminate choice types that are equivalent to monotypes, or equivalently, remove all positions that do not contribute to type errors. Thus, there is no need to simplify types nested in a choice D in Figure 7. Also, we formally define the selection operation $\lfloor \phi \rfloor_s$ in Figure 7. Selection extends naturally to lists of selectors in the following way: $\lfloor \phi \rfloor_{s'\overline{s}} = \lfloor \lfloor \phi \rfloor_{s'} \rfloor_{\overline{s}}$.

Next we want to establish the correctness of the inferred type changes. Formally, a *type update* is a mapping from program locations to monotypes. The intended meaning of one particular type update $l \mapsto \tau$ is to change the expression at l to an expression of type τ. We use δ to range over type updates. A type update is given by the locations and the second component of the corresponding choice types in the choice environment. We use $\downarrow \cdot$ to extract that mapping from Δ. The definition is $\downarrow \Delta = \{l \mapsto \tau_2 \mid (l, D\langle \tau_1, \tau_2 \rangle) \in \Delta\}$. (For the time being we assume that all the alternatives of choices in Δ are monotypes; we will lift this restriction later.) For example, with $\Delta = \{(l, A\langle \texttt{Int}, \texttt{Bool} \rangle)\}$ we have $\downarrow \Delta = \{l \mapsto \texttt{Bool}\}$.

The application of a type update is part of a *type update system* that is defined by the set of typing rules shown in Figure 8. These typing rules are identical to an ordinaryHM type system, except that they allow to "override" the types of AST leaves according to a type update δ that is a parameter for the rules. We only show the rules for constants, variables, and applications since those for abstractions and let-expressions are obtained from the HM ones in the same way as the application rule by simply adding the δ parameter. We write more shortly $\delta(e)$ for $\delta(\ell(e))$, and

$$\textsc{Con-C}\ \frac{c \text{ is of type } \gamma}{\Gamma;\delta \vdash^C c : \delta(c)||\gamma} \qquad \textsc{Var-C}\ \Gamma;\delta \vdash^C x : \delta(x)||\{\overline{\alpha \mapsto \tau}\}(\Gamma(x)) \qquad \textsc{App-C}\ \frac{\Gamma;\delta \vdash^C e_1 : \tau_1 \to \tau \qquad \Gamma;\delta \vdash^C e_2 : \tau_1}{\Gamma;\delta \vdash^C e_1\ e_2 : \tau}$$

Fig. 8. Rules for the type-update system.

we use the "orelse" notation $\delta(e)||\tau$ to pick the type $\delta(e)$ if $\delta(e)$ is defined and τ otherwise.

The rules Con-C and Var-C employ a type update if it exists. Otherwise, the usual typing rules apply. Rule App-C delegates the application of change updates to subexpressions since we are considering change suggestions for AST leaves only.

We can now show that by applying any of the inferred type changes (using the rules in Figure 8), we obtain the same types that are encoded in the variational type potential computed by type-change inference. We employ the following additional notation. We write $\Delta.2$ for the list of selectors $D.2$ for each choice $D\langle\rangle$ in Δ. For example, $\{(\ell_1, A\langle \texttt{Int}, \texttt{Bool}\rangle), (\ell_2, B\langle \texttt{Bool}, \texttt{Int}\rangle)\}.2 = [A.2, B.2]$. We call a type environment that maps variables to plain types (types without choices) a *plain type environment*. Formally, we have the following result. (We assume that Δ has been simplified by $\Downarrow$ in Figure 7 and the alternatives of choices in Δ are plain (non-variational), as mentioned before.)

Theorem 1 (*Type-change inference is consistent*)

For any given e and plain environment Γ, if $\Gamma \vdash e : \phi|\Delta_1$, $\Delta_1 \Downarrow \Delta$, and $\lfloor\phi\rfloor_{\Delta.2} = \tau$, then $\Gamma;\downarrow\Delta \vdash^C e : \tau$.

This theorem makes two assumptions: (1) Γ is plain and (2) the alternatives of choices in Δ are plain. Neither assumptions implies a limitation of the typing relation $\vdash$. Assumption (1) is required because the typing relation $\vdash^C$ is only defined when Γ is plain. The reason is that $\vdash^C$ defines a type system that is similar to HM but also considers changes and HM is defined only for plain Γ. Assumption (2) helps to simplify the theorem and the proof process. Note that in this theorem, each typing derivation can produce exactly one type update for an ill-typed expression. For example, with $\Gamma \vdash \texttt{not 1} : A\langle\bot, \alpha_5\rangle|\Delta$ where $\Delta = \{(\ell(\texttt{not}), A\langle \texttt{Bool} \to \texttt{Bool}, \texttt{Int} \to \alpha_5\rangle)\}$, we can produce the type update that changes `not` to something of the type $\texttt{Int} \to \alpha_5$. For such a typing, both alternatives of choices in Δ will be plain. Of course, in general each typing derivation could encode multiple type updates, and the alternatives may be variational. In fact, Theorem 3 demonstrates that for each ill-typed expression there is one typing derivation that encode all possible type updates. Since the goal of this theorem is to prove that each type update encoded in the typing result of $\vdash$ is correct, we can, for now, assume that alternatives are plain. We lift this assumption from Lemma 3.

For the proof of this theorem, we need the following lemma, whose proof can be found in Chen *et al.* (2012).

Lemma 2

$$\begin{aligned}\lfloor\uparrow(\phi)\rfloor_s &= \uparrow(\lfloor\phi\rfloor_s)\\ \lfloor\pi \lhd \phi\rfloor_s &= \lfloor\pi\rfloor_s \lhd \lfloor\phi\rfloor_s\\ \lfloor\phi_1 \bowtie \phi_2\rfloor_s &= \lfloor\phi_1\rfloor_s \bowtie \lfloor\phi_2\rfloor_s\\ \lfloor\phi_1 \otimes \phi_2\rfloor_s &= \lfloor\phi_1\rfloor_s \otimes \lfloor\phi_2\rfloor_s\end{aligned}$$

Proof of Theorem 1

We consider two different cases. In the first case, $\Delta = \emptyset$. According to the definition of $\Downarrow$, this indicates that no changes have been made in the typing process for $\Gamma \vdash e : \phi|\Delta$. In this case, the assumption $\lfloor\phi\rfloor_{\Delta.2} = \tau$ simplifies to $\phi = \tau$. Thus, the theorem itself simplifies to $\Gamma \vdash e : \tau|\emptyset \Rightarrow \Gamma; \emptyset \vdash^C e : \tau$. This holds trivially since both type systems coincide with HM when no type changes have been inferred.

In the second case, $\Delta \neq \emptyset$. This indicates that type changes have been inferred during the typing process. We prove this case by structural induction over the typing relation defined in Figure 6.

Case Con: We have $e = c$ and $\Gamma \vdash c : D\langle\gamma,\tau\rangle|\{(\ell(c), D\langle\gamma,\tau\rangle)\}$ for a fresh choice D since alternatives of choices are plain. The type update in this case is

$$\delta = \downarrow\Delta = \downarrow\{(\ell(c), D\langle\gamma,\tau\rangle)\} = \{c \mapsto \tau\}.$$

Our goal is to prove $\Gamma; \delta \vdash^C c : \tau$. Based on the structure of e, the only rule that applies in Figure 8 is Con-C. The proof follows directly as δ changes c to τ.

Case Var: The proof for this case is very similar to that for case Con and is omitted here. Note that the instantiation of ϕ_1 in rule Var is irrelevant as the instantiation is overridden by the change.

Case App: We need to show that

$$\Gamma \vdash e_1\ e_2 : \phi|\Delta \Rightarrow \Gamma; \downarrow\Delta \vdash^C e_1\ e_2 : \lfloor\phi\rfloor_{\Delta.2}$$

with the following induction hypotheses:

$$\begin{aligned}\Gamma \vdash e_1 : \phi_1|\Delta_1 &\Rightarrow \Gamma; \downarrow\Delta_1 \vdash^C e_1 : \lfloor\phi_1\rfloor_{\Delta_1.2}\\ \Gamma \vdash e_2 : \phi_2|\Delta_2 &\Rightarrow \Gamma; \downarrow\Delta_2 \vdash^C e_2 : \lfloor\phi_2\rfloor_{\Delta_2.2}\end{aligned}$$

Additionally, ϕ is computed by ϕ_1 and ϕ_2 through a use of the App rule. Let $\uparrow(\phi_1) = \phi_{1l} \to \phi_{1r}$, then we have the following relation:

$$\begin{aligned}\tau &= \lfloor\phi\rfloor_{\Delta.2}\\ &= \lfloor\phi_{1l} \bowtie \phi_2 \lhd \phi_{1r}\rfloor_{\Delta.2}\\ &= \lfloor\phi_{1l}\rfloor_{\Delta.2} \bowtie \lfloor\phi_2\rfloor_{\Delta.2} \lhd \lfloor\phi_{1r}\rfloor_{\Delta.2} \qquad \text{By Lemma 2}\end{aligned}$$

According to the definitions of $\bowtie$ and $\lhd$, we have

$$\lfloor\phi_{1l}\rfloor_{\Delta.2} = \lfloor\phi_2\rfloor_{\Delta.2} \tag{1}$$

$$\lfloor\phi_{1r}\rfloor_{\Delta.2} = \tau. \tag{2}$$

Combing the fact that $\uparrow(\phi_1) = \phi_{1l} \to \phi_{1r}$ with the Equations (1) and (2), we have $\lfloor\uparrow(\phi_1)\rfloor_{\Delta.2} = \lfloor\phi_2\rfloor_{\Delta.2} \to \tau$. Based on the definition of $\uparrow$, we have the

following relation.

$$\lfloor \phi_1 \rfloor_{\Delta.2} = \lfloor \phi_2 \rfloor_{\Delta.2} \to \tau \tag{3}$$

From $\Gamma; \downarrow\Delta_1 \vdash^C e_1 : \lfloor \phi_1 \rfloor_{\Delta_1.2}$, we have $\Gamma; \downarrow\Delta \vdash^C e_1 : \lfloor \phi_1 \rfloor_{\Delta_1.2}$ because compared to Δ_1, Δ contains additional information that only has an impact on typing e_2. From $\Gamma; \downarrow\Delta \vdash^C e_1 : \lfloor \phi_1 \rfloor_{\Delta_1.2}$, we have $\Gamma; \downarrow\Delta \vdash^C e_1 : \lfloor \phi_1 \rfloor_{\Delta.2}$ because $\lfloor \phi_1 \rfloor_{\Delta_1.2}$ already yields a plain type, and expanding the decision $\Delta_1.2$ to $\Delta.2$ will not change the result. Overall, we have

$$\Gamma; \downarrow\Delta_1 \vdash^C e_1 : \lfloor \phi_1 \rfloor_{\Delta_1.2} \Rightarrow \Gamma; \downarrow\Delta \vdash^C e_1 : \lfloor \phi_1 \rfloor_{\Delta.2} \tag{4}$$

Similarly, we have

$$\Gamma; \downarrow\Delta_2 \vdash^C e_2 : \lfloor \phi_2 \rfloor_{\Delta_2.2} \Rightarrow \Gamma; \downarrow\Delta \vdash^C e_2 : \lfloor \phi_2 \rfloor_{\Delta.2} \tag{5}$$

From Equations (3) and (4), we have

$$\Gamma; \downarrow\Delta_1 \vdash^C e_1 : \lfloor \phi_1 \rfloor_{\Delta_1.2} \Rightarrow \Gamma; \downarrow\Delta \vdash^C e_1 : \lfloor \phi_2 \rfloor_{\Delta.2} \to \tau \tag{6}$$

From Equations (5) and (6), the typing rule App-C, and the induction hypotheses, we have $\Gamma; \downarrow\Delta \vdash^C e_1\ e_2 : \tau$. Since $\tau = \lfloor \phi \rfloor_{\Delta.2}$, we have $\Gamma; \downarrow\Delta \vdash^C e_1\ e_2 : \lfloor \phi \rfloor_{\Delta.2}$, completing the proof.

The case for the rule Abs is straightforward and those for rules Let and If are similar to that for rule App and are therefore omitted here. □

Moreover, type-change inference is complete, since it can generate a set of type changes for any desired type.

Theorem 2 (*Type-change inference is complete*)
For any e, Γ and δ, if $\Gamma; \delta \vdash^C e : \tau$, then there exist ϕ, Δ, and a typing derivation for $\Gamma \vdash e : \phi | \Delta$ such that $\downarrow\Delta = \delta$ and $\lfloor \phi \rfloor_{\Delta.2} = \tau$.

Proof
Again, we consider two cases. In the first case, $\delta = \varnothing$, which means no changes have been applied. Thus, $\Gamma; \varnothing \vdash^C e : \tau$ implies that the expression e under Γ is well typed. When typing using the rules in Figure 6, we make the second alternative the same as the first alternative for all choices. In other words, we create idempotent choices only. The theorem holds trivially in this case.

In the second case, $\delta \neq \varnothing$. We prove the lemma by structural induction over the typing relation in Figure 8.

Case Con-C: We have $e = c$. Since $\delta \neq \varnothing$, it must be of the form $\delta = \{c \mapsto \tau\}$. We construct the typing relation as $\Gamma \vdash c : D\langle \gamma, \tau \rangle | \{(\ell(c), D\langle \gamma, \tau \rangle)\}$, where γ is the type of c and D is a fresh choice. It's easy to verify that $\downarrow\{(\ell(c), D\langle \gamma, \tau \rangle)\} = \{c \mapsto \tau\}$ and $\lfloor D\langle \gamma, \tau \rangle \rfloor_{D.2} = \tau$.

Case Var-C: The proof of this case is the same as for Con-C and is omitted here. The only difference is that we are dealing with a variable reference rather than a constant.

Case App-C: We need to show that

$$\Gamma;\delta \vdash^C e_1\ e_2 : \tau \Rightarrow \Gamma \vdash e_1\ e_2 : \phi|\Delta \text{ with } \downarrow\Delta = \delta \text{ and } \lfloor\phi\rfloor_{\Delta.2} = \tau$$

with the following induction hypotheses:

$$\Gamma;\delta \vdash^C e_1 : \tau_1 \to \tau \ \Rightarrow\ \Gamma \vdash e_1 : \phi_1|\Delta \text{ with } \downarrow\Delta = \delta \text{ and } \lfloor\phi_1\rfloor_{\Delta.2} = \tau_1 \to \tau$$
$$\Gamma;\delta \vdash^C e_2 : \tau_1 \ \Rightarrow\ \Gamma \vdash e_2 : \phi_2|\Delta \text{ with } \downarrow\Delta = \delta \text{ and } \lfloor\phi_2\rfloor_{\Delta.2} = \tau_1$$

We can split Δ into Δ_1 and Δ_2 such that they contain the change information for e_1 and e_2, respectively. With that, we have the following relations:

$$\Gamma;\delta \vdash^C e_1 : \tau_1 \to \tau \ \Rightarrow\ \Gamma \vdash e_1 : \phi_1|\Delta_1 \text{ with } \lfloor\phi_1\rfloor_{\Delta_1.2} = \tau_1 \to \tau$$
$$\Gamma;\delta \vdash^C e_2 : \tau_1 \ \Rightarrow\ \Gamma \vdash e_2 : \phi_2|\Delta_2 \text{ with } \lfloor\phi_2\rfloor_{\Delta_2.2} = \tau_1$$

Since $\Delta_1.2 \subseteq \Delta.2$ and $\lfloor\phi_1\rfloor_{\Delta_1.2} = \tau_1 \to \tau$, we have $\lfloor\phi_1\rfloor_{\Delta.2} = \tau_1 \to \tau$. We have a similar result regarding ϕ_2. Based on these, we arrive at the following relations:

$$\Gamma;\delta \vdash^C e_1 : \tau_1 \to \tau \ \Rightarrow\ \Gamma \vdash e_1 : \phi_1|\Delta_1 \text{ with } \lfloor\phi_1\rfloor_{\Delta.2} = \tau_1 \to \tau$$
$$\Gamma;\delta \vdash^C e_2 : \tau_1 \ \Rightarrow\ \Gamma \vdash e_2 : \phi_2|\Delta_2 \text{ with } \lfloor\phi_2\rfloor_{\Delta.2} = \tau_1$$

Let $\uparrow(\phi_1) = \phi_{1l} \to \phi_{1r}$. We compute $\lfloor\phi\rfloor_{\Delta.2}$ as follows:

$$\begin{aligned}
\lfloor\phi\rfloor_{\Delta.2} &= \lfloor\phi_{1l} \bowtie \phi_2 \lhd \phi_{1r}\rfloor_{\Delta.2} \\
&= \lfloor\phi_{1l}\rfloor_{\Delta.2} \bowtie \lfloor\phi_2\rfloor_{\Delta.2} \lhd \lfloor\phi_{1r}\rfloor_{\Delta.2} && \text{By Lemma 2} \\
&= \tau_1 \bowtie \tau_1 \lhd \tau && \text{By relation between } \lfloor\phi_1\rfloor_{\Delta.2}, \phi_{1l}, \text{and } \phi_{1r} \\
&= \top \lhd \tau \\
&= \tau
\end{aligned}$$

This shows that $\lfloor\phi\rfloor_{\Delta.2} = \tau$. Based on the induction hypotheses, $\downarrow\Delta = \delta$, which completes the proof.

We can prove the cases for other rules in Figure 8 similarly. □

The introduction of arbitrary alternative types in rules Con, Var, and Unbound are the reason that type-change inference is highly non-deterministic, that is, for any expression e we can generate an arbitrary number of type derivations with different type potentials and corresponding type changes.

Many of those derivations do not make much sense. For example, we can derive $\Gamma \vdash \texttt{5} : A\langle\texttt{Int}, \texttt{Bool}\rangle|\Delta$ where $\Delta = \{(\ell_{\texttt{5}}(\texttt{5}), A\langle\texttt{Int}, \texttt{Bool}\rangle)\}$. However, since the expression `5` is type correct, it does not make sense to suggest a change for it.

On the other hand, the ill-typed expression $e =$ `not (succ 5)` can be typed in two different ways that can correct the error, yielding two different type potentials and type changes. We can either suggest to change `not` to an expression of type `Int` $\to \alpha_1$, or we can suggest to change `succ` into something of type `Int` $\to$ `Bool`. The first suggestion is obtained by a derivation for $\Gamma \vdash e : A\langle\bot, \alpha_1\rangle|\Delta_1$ with $\Delta_1 = \{(\ell_e(\texttt{not}), A\langle\texttt{Bool} \to \texttt{Bool}, \texttt{Int} \to \alpha_1\rangle)\}$. The second suggestion is obtained by a derivation for $\Gamma \vdash e : B\langle\bot, \texttt{Bool}\rangle|\Delta_2$ with $\Delta_2 = \{(\ell_e(\texttt{succ}), B\langle\texttt{Int} \to \texttt{Int}, \texttt{Int} \to \texttt{Bool}\rangle)\}$.

Interestingly, we can combine both suggestions by deriving a more general typing statement, that is, we can derive the judgment $\Gamma \vdash e : A\langle B\langle \bot, \texttt{Bool}\rangle, B\langle \alpha_1, \alpha_2\rangle\rangle | \Delta_3$ where

$$\Delta_3 = \{(\ell_e(\texttt{not}), A\langle \texttt{Bool} \to \texttt{Bool}, B\langle \texttt{Int} \to \alpha_1, \alpha_3 \to \alpha_2\rangle\rangle), \\ (\ell_e(\texttt{succ}), B\langle \texttt{Int} \to \texttt{Int}, A\langle \texttt{Int} \to \texttt{Bool}, \texttt{Int} \to \alpha_3\rangle\rangle)\}$$

We can show that the third typing is better than the first two in the sense that its result type (a) contains fewer type errors than either of the result types and (b) is more general. For example, by selecting $[A.1, B.2]$ from both result types, we obtain $\bot$ and `Bool`, respectively. Making the same selection into the third result type, we obtain `Bool`. Likewise, when we select the result types of these three typings with $[A.2, B.1]$, we get the types α_1, $\bot$, and α_1, respectively. For each selection, the third result type is better than either one of the first two.

In the following, we show that this is not an accident, but that we can, in fact, always find a most general change suggestion from which all other suggestions can be instantiated.

First, we extend the function $\downarrow$ to take as an additional parameter a list of selectors $\bar{s}$. We also extend the definition to work with general variational types (and not just monotypes).

$$\downarrow_{\bar{s}}\Delta = \{l \mapsto \lfloor \phi_2 \rfloor_{\bar{s}} \mid (l, D\langle \phi_1, \phi_2\rangle) \in \Delta \wedge D.2 \in \bar{s}\}$$

Intuitively, we consider all the locations for which the second alternative of the corresponding choices are chosen. We need to apply the selection $\lfloor \phi_2 \rfloor_{\bar{s}}$ because each variational type may include other choice types that are subject to selection by $\bar{s}$.

Next we will show that type-change inference produces most general type changes from which any individual type change can be instantiated. We observe that type potentials and type changes can be compared in principally two different ways. First, the result of type-change inference ϕ can be *more defined* than another result ϕ', which means that for any $\bar{s}$ for which $\lfloor \phi' \rfloor_{\bar{s}}$ yields a monotype, so does $\lfloor \phi \rfloor_{\bar{s}}$. Second, a result ϕ can be *more general* than another result ϕ', written as $\phi \leqslant \phi'$, if there is some type substitution η such that $\phi' = \eta(\phi)$. Similarly, we call a type update δ_1 more general than another type update δ_2, written as $\delta_1 \leqslant \delta_2$, if $dom(\delta_1) = dom(\delta_2)$ and there is some η such that for all l: $\delta_2(l) = \eta(\delta_1(l))$.

Since we have these two different relationships between type changes, we have to show the generality of type-change inference in several steps.

First, we show that we can generalize any type change that produces a type error in the resulting variational type for a particular selection when there is another type change that does not produce a type error for the same selection. In the following lemma, we stipulate that the two typings $\Gamma \vdash e : \phi_1 | \Delta_1$ and $\Gamma \vdash e : \phi_2 | \Delta_2$ assign the same choice name to the same program location.

Lemma 3 (*Most defined type changes*)
Given e and Γ and two typings $\Gamma \vdash e : \phi_1 | \Delta_1$ and $\Gamma \vdash e : \phi_2 | \Delta_2$, for any $\bar{s}$ if $\lfloor \phi_1 \rfloor_{\bar{s}} = \bot$ and $\lfloor \phi_2 \rfloor_{\bar{s}} = \tau$, then there is a typing $\Gamma \vdash e : \phi_3 | \Delta_3$ such that

- $\lfloor \phi_3 \rfloor_{\bar{s}} = \lfloor \phi_2 \rfloor_{\bar{s}}$ and for all other $\bar{s}'$ $\lfloor \phi_3 \rfloor_{\bar{s}'} = \lfloor \phi_1 \rfloor_{\bar{s}'}$.
- $\downarrow_{\bar{s}} \Delta_3 = \downarrow_{\bar{s}} \Delta_2$ and $\downarrow_{\bar{s}'} \Delta_3 = \downarrow_{\bar{s}'} \Delta_1$ for all other $\bar{s}'$.

Proof

We construct a typing $\Gamma \vdash e : \phi_3 | \Delta_3$ so that both conditions of the lemma are satisfied. To construct a typing using the rules in Figure 6, we need to designate the second alternatives of the freshly created choices and the mappings for instantiating type schemas. Once they are decided, the typing is determined for the given e and Γ. The construction is based on a structural induction over the typing relation in Figure 6. The general idea is that the third typing is the same as the second typing at $\bar{s}$ and is the same as the first typing at all other $\bar{s}'$s differing from $\bar{s}$. We can realize this because each node has a variational type that can be viewed as a tree with choices as internal nodes and plain types as leaves. Moreover, we can replace one branch of the tree without affecting other branches. Specifically, we can first make the third typing the same as the first typing and then replace, for each type in the third typing, the branch identified by $\bar{s}$ with the same branch of the corresponding type from the second typing. The proof may be best understood by also reading the example after this proof at the same time. We prove a stronger lemma by dropping the conditions that $\lfloor \phi_1 \rfloor_{\bar{s}} = \bot$ and $\lfloor \phi_2 \rfloor_{\bar{s}} = \tau$.

Case Con: We need to further consider two sub-cases. Assume $\phi_1 = D\langle \gamma, \phi_1' \rangle$ and $\phi_2 = D\langle \gamma, \phi_2' \rangle$, where γ is the type of the constant c. In this case, ϕ_1 and ϕ_2 determine the contents of Δ_1 and Δ_2, respectively.

(1) $D.2 \in \bar{s}$. Let $\phi_3 = D\langle \gamma, \phi_3' \rangle$ where $\phi_3' = expand(\bar{s}, \phi_1', \phi_2')$. Here $expand(\bar{s}, \phi_1', \phi_2')$ builds a type ϕ such that $\lfloor \phi \rfloor_{\bar{s}} = \phi_2'$ and $\lfloor \phi \rfloor_{\bar{s}'} = \phi_1'$ for all other $\bar{s}'$. This function is formally defined as follows:

$$
\begin{aligned}
expand(D.1\bar{s}, \phi_1', \phi_2') &= D\langle expand(\bar{s}, \phi_1', \phi_2'), \phi_1' \rangle \\
expand(D.2\bar{s}, \phi_1', \phi_2') &= D\langle \phi_1', expand(\bar{s}, \phi_1', \phi_2') \rangle \\
expand(D.1, \phi_1', \phi_2') &= D\langle \phi_2', \phi_1' \rangle \\
expand(D.2, \phi_1', \phi_2') &= D\langle \phi_1', \phi_2' \rangle
\end{aligned}
$$

Given a decision, we write $s\bar{s}$ to single out an arbitrary selector s from that decision and bind the remaining to $\bar{s}$.

With the constructed ϕ_3 we can verify that $\lfloor \phi_3 \rfloor_{\bar{s}} = \lfloor \phi_3' \rfloor_{\bar{s}} = \phi_2' = \lfloor \phi_2 \rfloor_{\bar{s}}$ and $\lfloor \phi_3 \rfloor_{\bar{s}'} = \lfloor \phi_3' \rfloor_{\bar{s}'} = \phi_1' = \lfloor \phi_1 \rfloor_{\bar{s}'}$ for all other $\bar{s}'$. Since ϕ_3 determines Δ_3, verifying the second condition of the lemma follows directly from that of the first condition.

(2) $D.2 \notin \bar{s}$. Let $\phi_3 = \phi_1$. When $D.2 \notin \bar{s}$, we have both $\lfloor \phi_1 \rfloor_{\bar{s}} = \gamma$ and $\lfloor \phi_2 \rfloor_{\bar{s}} = \gamma$, which means that the change does not affect the decision $\bar{s}$. Thus, we do not need to change the type of c. We can verify that $\lfloor \phi_3 \rfloor_{\bar{s}} = c = \lfloor \phi_2 \rfloor_{\bar{s}}$ and $\lfloor \phi_3 \rfloor_{\bar{s}'} = c = \lfloor \phi_1 \rfloor_{\bar{s}'}$ for all other $\bar{s}'$.

Case Var: The proof is similar to that of case Con and is omitted here.

Case App: For this case, we do not need to construct anything but only have to show that the lemma is preserved over applying the App rule. The induction hypotheses are as follows:

$$\begin{array}{lll}
\Gamma \vdash e_1 : \phi_{11}|\Delta_{11} & \Gamma \vdash e_2 : \phi_{12}|\Delta_{12} & \Gamma \vdash e_1\ e_2 : \phi_1|\Delta_1 \\
\Gamma \vdash e_1 : \phi_{21}|\Delta_{21} & \Gamma \vdash e_2 : \phi_{22}|\Delta_{22} & \Gamma \vdash e_1\ e_2 : \phi_2|\Delta_2 \\
\Gamma \vdash e_1 : \phi_{31}|\Delta_{31} & \Gamma \vdash e_2 : \phi_{32}|\Delta_{32} & \Gamma \vdash e_1\ e_2 : \phi_3|\Delta_3 \\
\lfloor\phi_{31}\rfloor_{\bar{s}} = \lfloor\phi_{21}\rfloor_{\bar{s}} & \lfloor\phi_{31}\rfloor_{\bar{s}'} = \lfloor\phi_{11}\rfloor_{\bar{s}'} & \downarrow_{\bar{s}}\Delta_{31} = \downarrow_{\bar{s}}\Delta_{21} \quad \downarrow_{\bar{s}'}\Delta_{31} = \downarrow_{\bar{s}'}\Delta_{11} \\
\lfloor\phi_{32}\rfloor_{\bar{s}} = \lfloor\phi_{22}\rfloor_{\bar{s}} & \lfloor\phi_{32}\rfloor_{\bar{s}'} = \lfloor\phi_{12}\rfloor_{\bar{s}'} & \downarrow_{\bar{s}}\Delta_{32} = \downarrow_{\bar{s}}\Delta_{22} \quad \downarrow_{\bar{s}'}\Delta_{32} = \downarrow_{\bar{s}'}\Delta_{12}
\end{array}$$

We need to show that

$$\lfloor\phi_3\rfloor_{\bar{s}} = \lfloor\phi_2\rfloor_{\bar{s}} \quad \lfloor\phi_3\rfloor_{\bar{s}'} = \lfloor\phi_1\rfloor_{\bar{s}'} \quad \downarrow_{\bar{s}}\Delta_3 = \downarrow_{\bar{s}}\Delta_2 \quad \downarrow_{\bar{s}'}\Delta_3 = \downarrow_{\bar{s}'}\Delta_1$$

In the following, we show $\lfloor\phi_3\rfloor_{\bar{s}} = \lfloor\phi_2\rfloor_{\bar{s}}$ with the assumptions that $\uparrow(\phi_{31}) = \phi_{31l} \to \phi_{31r}$ and $\uparrow(\phi_{21}) = \phi_{21l} \to \phi_{21r}$.

$$\begin{aligned}
\lfloor\phi_3\rfloor_{\bar{s}} &= \lfloor\phi_{31l} \bowtie \phi_{32} \lhd \phi_{31r}\rfloor_{\bar{s}} \\
&= \lfloor\phi_{31l}\rfloor_{\bar{s}} \bowtie \lfloor\phi_{32}\rfloor_{\bar{s}} \lhd \lfloor\phi_{31r}\rfloor_{\bar{s}} && \text{By Lemma 2} \\
&= \lfloor\phi_{31l}\rfloor_{\bar{s}} \bowtie \lfloor\phi_{22}\rfloor_{\bar{s}} \lhd \lfloor\phi_{31r}\rfloor_{\bar{s}} && \text{By induction hypothesis} \\
&= \lfloor\phi_{21l}\rfloor_{\bar{s}} \bowtie \lfloor\phi_{22}\rfloor_{\bar{s}} \lhd \lfloor\phi_{21r}\rfloor_{\bar{s}} && \text{By Lemma 2 and induction hypothesis} \\
&= \lfloor\phi_{21l} \bowtie \phi_{22} \lhd \phi_{21r}\rfloor_{\bar{s}} && \text{By Lemma 2} \\
&= \lfloor\phi_2\rfloor_{\bar{s}}
\end{aligned}$$

We can prove $\lfloor\phi_3\rfloor_{\bar{s}'} = \lfloor\phi_1\rfloor_{\bar{s}'}$ similarly. Since $\Delta_3 = \Delta_{31} \cup \Delta_{32} = \Delta_{21} \cup \Delta_{22} = \Delta_2$, we have $\downarrow_{\bar{s}}\Delta_3 = \downarrow_{\bar{s}}\Delta_2$.

The proofs for other cases are very similar to the case App and are omitted here. □

We use an example to illustrate the proof process. We consider constructing the new typing for the example expression $e =$ `not (succ 5)` under the following typings:

$$\begin{array}{lll}
\Gamma \vdash e : \phi_1|\Delta_1 & \phi_1 = A\langle\bot, \alpha_1\rangle & \Delta_1 = \{(\ell(\texttt{not}), A\langle\texttt{Bool} \to \texttt{Bool}, \texttt{Int} \to \alpha_1\rangle) \\
 & & \quad (\ell(\texttt{succ}), B\langle\texttt{Int} \to \texttt{Int}, \texttt{Int} \to \texttt{Int}\rangle)\} \\
\Gamma \vdash e : \phi_2|\Delta_2 & \phi_2 = B\langle\bot, \texttt{Bool}\rangle & \Delta_2 = \{(\ell(\texttt{not}), A\langle\texttt{Bool} \to \texttt{Bool}, \texttt{Bool} \to \texttt{Bool}\rangle) \\
 & & \quad (\ell(\texttt{succ}), B\langle\texttt{Int} \to \texttt{Int}, \texttt{Int} \to \texttt{Bool}\rangle)\}
\end{array}$$

We consider $\bar{s} = [A.1, B.2]$ and observe that $\lfloor\phi_1\rfloor_{\bar{s}} = \bot$ and $\lfloor\phi_2\rfloor_{\bar{s}} = \texttt{Bool}$. We construct $\Gamma \vdash e : \phi_3|\Delta_3$ as follows. For `not`, the choice created is A. Since $A.2 \notin [A.1, B.2]$, the type for `not` is $A\langle\texttt{Bool} \to \texttt{Bool}, \texttt{Int} \to \alpha_1\rangle$, the type for `not` in Δ_1. For `succ`, the created choice is B. Since $B.2 \in [A.1, B.2]$, the type of `succ`, written as $\phi_{\texttt{succ}}$, can be computed as follows:

$$\begin{aligned}
\phi_{\texttt{succ}} &= B\langle\texttt{Int} \to \texttt{Int}, \mathit{expand}([A.1, B.2], \texttt{Int} \to \texttt{Int}, \texttt{Int} \to \texttt{Bool})\rangle \\
&= B\langle\texttt{Int} \to \texttt{Int}, A\langle\mathit{expand}(B.2, \texttt{Int} \to \texttt{Int}, \texttt{Int} \to \texttt{Bool}), \texttt{Int} \to \texttt{Int}\rangle\rangle \\
&= B\langle\texttt{Int} \to \texttt{Int}, A\langle B\langle\texttt{Int} \to \texttt{Int}, \texttt{Int} \to \texttt{Bool}\rangle, \texttt{Int} \to \texttt{Int}\rangle\rangle \\
&= B\langle\texttt{Int} \to \texttt{Int}, A\langle\texttt{Int} \to \texttt{Bool}, \texttt{Int} \to \texttt{Int}\rangle\rangle
\end{aligned}$$

The type for `5` is always `Int`. Now that we have specified types for `not`, `succ`, and `5`, we compute $\phi_3 = A\langle B\langle\bot, \texttt{Bool}\rangle, \alpha_1\rangle$. The content for Δ_3 is easy to construct, and we omit it here. It is easy to verify that $\lfloor\phi_3\rfloor_{[A.1,B.2]} = \texttt{Bool} = \lfloor\phi_2\rfloor_{[A.1,B.2]}$ and that

for all other $\bar{s}'$ we have $\lfloor\phi_3\rfloor_{\bar{s}'} = \lfloor\phi_1\rfloor_{\bar{s}'}$. We can verify that the relation given in the lemma holds for Δ_1, Δ_2, and Δ_3.

Next we show that given any two type changes, we can always find a type change that generalizes the two.

Lemma 4 (*Generalizability of type changes*)
For any two typings $\Gamma \vdash e : \phi_1|\Delta_1$ and $\Gamma \vdash e : \phi_2|\Delta_2$ and any $\bar{s}$, there is a typing $\Gamma \vdash e : \phi_3|\Delta_3$ such that

- $\lfloor\phi_3\rfloor_{\bar{s}} \leqslant \lfloor\phi_1\rfloor_{\bar{s}}$, $\lfloor\phi_3\rfloor_{\bar{s}} \leqslant \lfloor\phi_2\rfloor_{\bar{s}}$ and for all other $\bar{s}'$, $\lfloor\phi_3\rfloor_{\bar{s}'} = \lfloor\phi_1\rfloor_{\bar{s}'}$.
- $\downarrow_{\bar{s}}\Delta_3 \leqslant \downarrow_{\bar{s}}\Delta_1$, $\downarrow_{\bar{s}}\Delta_3 \leqslant \downarrow_{\bar{s}}\Delta_2$ and for all other $\bar{s}'$, $\downarrow_{\bar{s}'}\Delta_3 = \downarrow_{\bar{s}'}\Delta_1$.

The proof strategy is similar to that for Lemma 3, except for one subtle difference. In proving Lemma 3, we take something directly from the second typing and merge it into the first to get the third without any changes. This strategy is insufficient here. We use two examples to illustrate the problem. Both examples relate to the typing of expression `not 1`.

In the first example, we remove the type error with the following potential typings:

$$\begin{array}{llll}
\Gamma \vdash \texttt{not 1} : \phi_1|\Delta_1 & \phi_1 = A\langle\bot, \texttt{Int}\rangle & \Delta_1 = \{(\ell(\texttt{not}), A\langle \texttt{Bool} \to \texttt{Bool}, \texttt{Int} \to \texttt{Int}\rangle)\} \\
\Gamma \vdash \texttt{not 1} : \phi_2|\Delta_2 & \phi_2 = A\langle\bot, \texttt{Bool}\rangle & \Delta_2 = \{(\ell(\texttt{not}), A\langle \texttt{Bool} \to \texttt{Bool}, \texttt{Int} \to \texttt{Bool}\rangle)\}
\end{array}$$

We observe that these typings have different result types and that neither is more general than the other. We know that there must exist a third typing that gives a more general result type. The type that is more general than both `Int` and `Bool` is a type variable, say α. We can achieve this result type by changing `not` to something of type $\texttt{Int} \to \alpha$. This change can be derived by looking at the types that `not` is changed to in both typings. The type of `not` is changed to $\texttt{Int} \to \texttt{Int}$ and $\texttt{Int} \to \texttt{Bool}$, respectively. A type that is more general than the both is $\texttt{Int} \to \alpha$.

In general, however, we need to accommodate the impact of changing the type for some subexpression on the typing of the whole expression. In the second example, we use the following typings to remove the type error in `not 1`:

$$\begin{array}{lll}
\Gamma \vdash \texttt{not 1} : \phi_1|\Delta_1 & \phi_1 = A\langle\bot, \texttt{Int}\rangle & \Delta_1 = \{(\ell(\texttt{not}), A\langle \texttt{Bool} \to \texttt{Bool}, \texttt{Int} \to \texttt{Int}\rangle), \\
 & & \quad (\ell(\texttt{1}), B\langle \texttt{Int}, \texttt{Int}\rangle)\} \\
\Gamma \vdash \texttt{not 1} : \phi_2|\Delta_2 & \phi_2 = B\langle\bot, \texttt{Bool}\rangle & \Delta_2 = \{(\ell(\texttt{not}), A\langle \texttt{Bool} \to \texttt{Bool}, \texttt{Bool} \to \texttt{Bool}\rangle) \\
 & & \quad (\ell(\texttt{1}), B\langle \texttt{Int}, \texttt{Bool}\rangle)\}
\end{array}$$

We consider $\bar{s} = [A.2, B.2]$ and observe that $\lfloor\phi_1\rfloor_{\bar{s}} = \texttt{Int}$ and $\lfloor\phi_2\rfloor_{\bar{s}} = \texttt{Bool}$. Now in order to get a more general typing for the expression with α being the result type, we need to change the types assigned to subexpressions `not` and `1`. First, how should we change the type for `not`? Since these two typings assign it $\texttt{Int} \to \texttt{Int}$ and $\texttt{Bool} \to \texttt{Bool}$, respectively, a more general type is of the form $\alpha_1 \to \alpha$, following the idea of anti-unification (Pfenning, 1991). In other words, we assign $\alpha_1 \to \alpha$ to `not`. Now how about the type for `1`? The two typings change it to `Int` and `Bool`, respectively. We may be tempted to assign it an arbitrarily more general type, for example α_3. However, this change will make `not 1` ill typed because the domain type

of the function, α_1, does not match the type of the argument, α_3. We should instead assign `1` the type α_1, taking the fact that `not` has changed to $\alpha_1 \to \alpha$ into account.

In summary, while we need to generalize the types for certain subexpressions during the construction process, we should perform it consistently among all subexpressions. To simplify the presentation, we assume that there exists a function $postgen_e(l, \tau)$ that returns the type for the location l in e after generalization. We usually omit the subscript when the context makes it clear what e is. When no constraints have been seen so far for l in e, $postgen_e(l, \tau)$ returns a type that has the same structure as τ except that primitive types are replaced by fresh type variables. Otherwise, it returns the type that satisfies the type constraints among subexpressions. For example, consider again $e =$ `not 1`. For e, we have $postgen(\ell(\texttt{not}), \texttt{Int} \to \texttt{Int}) = \alpha_1 \to \alpha_2$. Since we have not seen any constraints for `not`, we assign a fresh type variable to each `Int`. For `1`, we need to consider the constraint between `not` and `1`, and we have $postgen(\ell(\texttt{1}), \texttt{Int}) = \alpha_1$. In static typing, the constraints among all subexpressions are easy to derive. Thus, the function $postgen_e(l, \tau)$ is easy to compute, and the definition is omitted here.

Proof of Lemma 4

The proof is by constructing a new typing based on the given typings so that both conditions of the lemma are satisfied.

Case Con: Assume $e = c$, $\phi_1 = D\langle\gamma, \phi_1'\rangle$, and $\phi_2 = D\langle\gamma, \phi_2'\rangle$, where γ is the type of c. We further consider two sub-cases.

(a) $D.2 \in \bar{s}$. Let $\phi_3 = D\langle\gamma, expand(\bar{s}, \phi_1', postgen(\ell(c)))\rangle$ and $\Delta_3 = \{(\ell(c), \phi_3)\}$. We can easily verify that both conditions of the lemma hold.

(b) $D.2 \notin \bar{s}$. Let $\phi_3 = \phi_1$ and $\Delta_3 = \Delta_1$. We simply do not make any change because changing c will not affect the result selected with $\bar{s}$.

Case Var: The proof is similar to that for Con and is omitted here.

Case App: We show the proof for the first condition about the relation among ϕ_1, ϕ_2, and ϕ_3. Since the proof about relations among Δ_1, Δ_2, and Δ_3 is almost the same as in proof for Lemma 3 and is rather simple, we omit it here. We have the following induction hypotheses.

$$\begin{array}{lll}
\Gamma \vdash e_1 : \phi_{11} | \Delta_{11} & \Gamma \vdash e_2 : \phi_{12} | \Delta_{12} & \Gamma \vdash e_1\ e_2 : \phi_1 | \Delta_1 \\
\Gamma \vdash e_1 : \phi_{21} | \Delta_{21} & \Gamma \vdash e_2 : \phi_{22} | \Delta_{22} & \Gamma \vdash e_1\ e_2 : \phi_2 | \Delta_2 \\
\Gamma \vdash e_1 : \phi_{31} | \Delta_{31} & \Gamma \vdash e_2 : \phi_{32} | \Delta_{32} & \Gamma \vdash e_1\ e_2 : \phi_3 | \Delta_3 \\
\lfloor \phi_{31} \rfloor_{\bar{s}} \leqslant \lfloor \phi_{11} \rfloor_{\bar{s}} & \lfloor \phi_{31} \rfloor_{\bar{s}} \leqslant \lfloor \phi_{21} \rfloor_{\bar{s}} & \lfloor \phi_{31} \rfloor_{\bar{s}'} = \lfloor \phi_{11} \rfloor_{\bar{s}'} \\
\lfloor \phi_{32} \rfloor_{\bar{s}} \leqslant \lfloor \phi_{12} \rfloor_{\bar{s}} & \lfloor \phi_{32} \rfloor_{\bar{s}} \leqslant \lfloor \phi_{22} \rfloor_{\bar{s}} & \lfloor \phi_{32} \rfloor_{\bar{s}'} = \lfloor \phi_{12} \rfloor_{\bar{s}'}
\end{array}$$

We need to show that

$$\lfloor \phi_3 \rfloor_{\bar{s}} \leqslant \lfloor \phi_1 \rfloor_{\bar{s}} \quad \lfloor \phi_3 \rfloor_{\bar{s}} \leqslant \lfloor \phi_2 \rfloor_{\bar{s}} \quad \lfloor \phi_3 \rfloor_{\bar{s}'} = \lfloor \phi_1 \rfloor_{\bar{s}'}$$

In the following, we show $\lfloor\phi_3\rfloor_{\bar{s}} \leqslant \lfloor\phi_1\rfloor_{\bar{s}}$ with the assumptions that $\uparrow(\phi_{31}) = \phi_{31l} \to \phi_{31r}$ and $\uparrow(\phi_{21}) = \phi_{21l} \to \phi_{21r}$.

$$\begin{aligned}
\lfloor\phi_3\rfloor_{\bar{s}} &= \lfloor\phi_{31l} \bowtie \phi_{32} \lhd \phi_{31r}\rfloor_{\bar{s}} \\
&= \lfloor\phi_{31l}\rfloor_{\bar{s}} \bowtie \lfloor\phi_{32}\rfloor_{\bar{s}} \lhd \lfloor\phi_{31r}\rfloor_{\bar{s}} && \text{By Lemma 2} \\
&= \top \lhd \lfloor\phi_{31r}\rfloor_{\bar{s}} && \text{By definition of } \textit{postgen} \\
&= \lfloor\phi_{31r}\rfloor_{\bar{s}} \\
&\leqslant \lfloor\phi_{11r}\rfloor_{\bar{s}} && \text{By induction hypothesis} \\
&= \lfloor\phi_1\rfloor_{\bar{s}} && \text{By a similar reasoning for } \phi_{11r}
\end{aligned}$$

We can prove $\lfloor\phi_3\rfloor_{\bar{s}} \leqslant \lfloor\phi_2\rfloor_{\bar{s}}$ and $\lfloor\phi_3\rfloor_{\bar{s}'} = \lfloor\phi_1\rfloor_{\bar{s}'}$ similarly.

The proof for other rules is similar to that for App and is omitted here. □

We can now combine and generalize Lemmas 3 and 4 and see that type-change inference can always produce maximally error-free and general results at the same time. This is an important result, captured in the following theorem.

Theorem 3 (*Most general and error-free type changes*)
Given e and Γ and two typings $\Gamma \vdash e : \phi_1|\Delta_1$ and $\Gamma \vdash e : \phi_2|\Delta_2$, there is a typing $\Gamma \vdash e : \phi_3|\Delta_3$ such that for any $\bar{s}$,

- if $\lfloor\phi_1\rfloor_{\bar{s}} = \bot$ and $\lfloor\phi_2\rfloor_{\bar{s}} = \tau$, then $\lfloor\phi_3\rfloor_{\bar{s}} = \tau$ and $\downarrow_{\bar{s}}\Delta_3 = \downarrow_{\bar{s}}\Delta_2$.
- if $\lfloor\phi_2\rfloor_{\bar{s}} = \bot$ and $\lfloor\phi_1\rfloor_{\bar{s}} = \tau$, then $\lfloor\phi_3\rfloor_{\bar{s}} = \tau$ and $\downarrow_{\bar{s}}\Delta_3 = \downarrow_{\bar{s}}\Delta_1$.
- if $\lfloor\phi_1\rfloor_{\bar{s}} = \tau_1$ and $\lfloor\phi_2\rfloor_{\bar{s}} = \tau_2$, then $\lfloor\phi_3\rfloor_{\bar{s}} \leqslant \tau_1$ and $\lfloor\phi_3\rfloor_{\bar{s}} \leqslant \tau_2$. Moreover, $\downarrow_{\bar{s}}\Delta_3 \leqslant \downarrow_{\bar{s}}\Delta_1$ and $\downarrow_{\bar{s}}\Delta_3 \leqslant \downarrow_{\bar{s}}\Delta_2$.

Proof
We delegate the actual construction process to the ones described in the proofs for Lemmas 3 and 4. In particular, given two typings, only one of the three cases mentioned in the theorem can occur. First, if $\lfloor\phi_1\rfloor_{\bar{s}} = \bot$ and $\lfloor\phi_2\rfloor_{\bar{s}} = \tau$, we use the idea presented in the proof for Lemma 3 to construct the new typing. The second case is a dual case of the first case, where $\lfloor\phi_2\rfloor_{\bar{s}} = \bot$ and $\lfloor\phi_1\rfloor_{\bar{s}} = \tau$. We proceed as we do in the first case but swap ϕ_1 and ϕ_2, and also Δ_1 and Δ_2. Finally, if $\lfloor\phi_1\rfloor_{\bar{s}} = \tau_1$ and $\lfloor\phi_2\rfloor_{\bar{s}} = \tau_2$, we use the construction process described in the proof for Lemma 4 to construct the new typing. In each case, the proof follows directly from Lemma 3 or Lemma 4. □

From Theorems 2 and 3 it follows that there is a typing for complete and principal type changes. We express this in the following theorem.

Theorem 4 (*Complete and principal type changes*)
Given e and plain Γ, there is a typing $\Gamma \vdash e : \phi|\Delta$ such that for any δ if $\Gamma; \delta \vdash^C e : \tau$, then there is some $\bar{s}$ such that $\lfloor\phi\rfloor_{\bar{s}} \leqslant \tau$ and $\downarrow_{\bar{s}}\Delta \leqslant \delta$.

Proof
Based on Theorem 2, if $\Gamma; \delta_i \vdash^C e : \tau_i$, then there is a typing and some $\bar{s}$ such that $\Gamma \vdash e : \phi_i|\Delta_i$ with $\lfloor\phi_i\rfloor_{\bar{s}} = \tau_i$ and $\downarrow_{\bar{s}}\Delta_i = \delta_i$. For different τ_is we may get different ϕ_is and Δ_is. Based on Theorem 3, there is a typing $\Gamma \vdash e : \phi|\Delta$ that is better than all typings with ϕ_is and Δ_is. The result holds. □

Finally, there is a close relationship between type-change inference and the HM type system. When type-change inference succeeds with an empty set of type changes, it produces a non-variational type that is identical to the one derived by HM. This result is captured in the following theorem, where we write $\Gamma \vdash^{HM} e : \tau$ to express that expression e has the type τ under Γ in the HM type system.

Theorem 5
For any given e and plain Γ, $\Gamma; \varnothing \vdash^{C} e : \tau \iff \Gamma \vdash^{HM} e : \tau$.

Proof
The proof is a straightforward induction over the typing relations in Figures 6 and 8. Note that when $\delta = \varnothing$, the typing rules in Figure 8 simplify exactly to those for the HM typing rules. □

Based on Theorem 1, Theorem 2, Theorem 5, and the fact that $\downarrow\varnothing = \varnothing$, we can infer that when a program is well typed, the type change-inference system and the HM system produce the same result.

Theorem 6
$\Gamma \vdash e : \tau|\varnothing$ if and only if $\Gamma \vdash^{HM} e : \tau$.

Proof
Based on Theorem 1, $\Gamma \vdash e : \tau|\varnothing$ implies that $\Gamma; \varnothing \vdash^{C} e : \tau$, which implies $\Gamma \vdash^{HM} e : \tau$ according to Theorem 5. Meanwhile, $\Gamma \vdash^{HM} e : \tau$ implies $\Gamma; \varnothing \vdash^{C} e : \tau$ according to Theorem 5. Based on Theorem 2, $\Gamma \vdash e : \tau|\varnothing$ holds. □

Note that $\Gamma \vdash e : \tau|\varnothing$ implies that $\Gamma \vdash e : \phi|\Delta$, $\phi \equiv \tau$, and $\Delta \Downarrow \varnothing$. This theorem also implies that type-change inference will never assign a monotype to a type-incorrect program.

5 A change inference algorithm

This section presents an algorithm for inferring type changes. We will discuss properties of the algorithm as well as strategies to bound its complexity.

Given the partial type unification algorithm presented in (Chen *et al.*, 2012), the inference algorithm is obtained by a straightforward translation of the typing rules presented in Figure 6. The cases for variable reference and `if` statements are shown in Figure 9. Function application is very similar to `if` statements, and the cases for abstractions and let-expressions can be derived from $\mathscr{W}$ by simply adding the threading of Δ.

For variable reference, the algorithm first tries to find the type of the variable in Γ and either instantiates the found type schema with fresh type variables or returns $\perp$ if the variable is unbound. After that, a fresh choice containing a fresh type variable is returned. The variable then has the returned choice type with the inferred type in the first alternative and the type variable in the second.

For typing `if` statements, we use an algorithm $\mathit{vunify}(\phi_1, \phi_2)$ for partial unification (Chen *et al.*, 2012) that generalizes standard unification to variational types.

$$
\begin{array}{l}
\mathit{infer} : \Gamma \times e \to \theta \times \phi \times \Delta \\
\mathit{infer}(\Gamma, x) = \\
\quad \phi' \leftarrow \mathit{inst}(\Gamma(x)) \qquad\qquad \text{– returns } \bot \text{ when } x \text{ is unbound} \\
\quad \phi \leftarrow D\langle \phi', \alpha \rangle \qquad\qquad \text{– } D \text{ and } \alpha \text{ are fresh} \\
\quad \text{return } (\varnothing, \phi, \{(\ell(x), \phi)\}) \\
\\
\mathit{infer}(\Gamma, \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3) = \\
\quad (\theta_1, \phi_1, \Delta_1) \leftarrow \mathit{infer}(\Gamma, e_1) \\
\quad (\theta', \pi') \leftarrow \mathit{vunify}(\phi_1, \texttt{Bool}) \\
\quad (\theta_2, \phi_2, \Delta_2) \leftarrow \mathit{infer}(\theta'\theta_1(\Gamma), e_2) \\
\quad (\theta_3, \phi_3, \Delta_3) \leftarrow \mathit{infer}(\theta_2\theta'\theta_1(\Gamma), e_3) \\
\quad (\theta_4, \pi_4) \leftarrow \mathit{vunify}(\theta_3(\phi_2), \phi_3,) \\
\quad \theta \leftarrow \theta_4\theta_3\theta_2\theta'\theta_1 \\
\quad \text{return } (\theta, \pi' \lhd (\pi_4 \lhd \theta_4(\phi_3)), \theta(\Delta_1 \cup \Delta_2 \cup \Delta_3))
\end{array}
$$

Fig. 9. An inference algorithm for variable references and conditionals.

We describe the algorithm briefly below. Otherwise, the algorithm follows in a straightforward way the usual strategy for type inference.

The algorithm *vunify* has two special aspects. First, it handles types that contain variations. Correspondingly, the returned unifier may map variables to variational types. For example, $\mathit{vunify}(\texttt{Int}, A\langle \texttt{Int}, \alpha \rangle)$ returns $\{\alpha \mapsto A\langle \alpha_2, \texttt{Int} \rangle\}$ as the unifier, where α_2 is a fresh type variable. (Intuitively, if we follow choice 1 of A, then α is irrelevant; only when we follow choice 2 of A, should α unify with `Int`. Therefore *vunify* maps α to a variational type to record this split.) Second, *vunify* also returns a typing pattern to indicate where unification fails and succeeds. For example, $\mathit{vunify}(\texttt{Int}, A\langle \texttt{Bool}, \alpha \rangle)$ returns $A\langle \bot, \top \rangle$, indicating that the unification fails in $A.1$ and succeeds in $A.2$. For any unification problem, *vunify* reconciles these two aspects such that for any unification problem it returns a most general unifier that also leads to as few $\bot$s as possible. The first aspect allows us to, for any expression, perform type inference once and generate all possible type updates. The second aspect allows us to solve unification problems in the presence of type errors, and thus infer types for ill-typed expressions. Section 3 presents a concrete example of using *vunify* for debugging the type error in `not 1`.

We can prove that the algorithm *infer* correctly implements the typing rules in Figure 6, as expressed in the following theorems.

Theorem 7 (*Type-change inference is sound*)
Given any e and Γ, if $\mathit{infer}(\Gamma, e) = (\theta, \phi, \Delta)$, then $\theta(\Gamma) \vdash e : \phi | \Delta$.

At the same time, the type inference is complete and principal. We use the auxiliary relation $\phi_1 \preceq \phi_2$ to express that for any $\bar{s}$, either $\lfloor \phi_2 \rfloor_{\bar{s}} = \bot$ or $\lfloor \phi_1 \rfloor_{\bar{s}} \leqslant \lfloor \phi_2 \rfloor_{\bar{s}}$. Intuitively, this expresses that either the corresponding variant in ϕ_1 is more general or more correct. We also define $\Delta_1 \preceq \Delta_2$ if for any $(l, \phi_1) \in \Delta_1$ and $(l, \phi_2) \in \Delta_2$ the condition $\phi_1 \preceq \phi_2$ holds.

Theorem 8 (*Type-change inference is complete and principal*)
If $\theta(\Gamma) \vdash e : \phi | \Delta$, then $\mathit{infer}(\Gamma, e) = (\theta_1, \phi_1, \Delta_1)$ such that $\theta = \eta_1\theta_1$ for some η_1, $\Delta_1 \preceq \Delta$, and $\phi_1 \preceq \phi$.

From Theorems 3 and 8, it follows that our type-change inference algorithm correctly computes all type changes for a given expression in one single run.

During the type-change inference process, choice types can become deeply nested, and the size of types can become exponential in the nesting levels. Fortunately, this can occur only for deeply nested function applications where each argument type is required to be the same. For example, the function $f : \alpha \to \alpha \to \ldots \to \alpha$ is more likely to cause this problem than the functions $g : \alpha_1 \to \alpha_2 \to \ldots \to \alpha_n$ and $h : \gamma_1 \to \gamma_2 \to \ldots \to \gamma_n$ because only the function f requires all argument types to be unified, which is the source of nested choices.

To illustrate, consider the expressions `and2 = 1 && True` and `equal2 = 1 == True`, where `(&&)` : `Bool` → `Bool` → `Bool` and `(==)` : $\alpha \to \alpha \to$ `Bool`. In each expression, the choices B and D are created for `1` and `True`, respectively. The choice A is created for `&&` in `and2` and for `==` in `equal2`. Let Δ_1 and Δ_2 be the change information (third return value) returned by *infer* for `and2` and `equal2`, respectively. We observe that $\Delta_1(\ell(1)) = B\langle \texttt{Int}, A\langle \texttt{Bool}, \alpha_1\rangle\rangle$, and $\Delta_2(\ell(1)) = B\langle \texttt{Int}, A\langle D\langle \texttt{Bool}, \alpha_4\rangle, \alpha_2\rangle\rangle$. The nesting of choices in $\Delta_2(\ell(1))$ is deeper than in $\Delta_1(\ell(1))$ because the argument types of `==` are both α and those of `&&` are constant types. This is also the case when we consider ℓ(`True`) in Δ_1 and Δ_2. The reason is that in `and2` the arguments can be changed independently of one another while in `equal2` the arguments have to be synchronized through the shared argument type α.

To keep the run-time complexity of our inference algorithm under control, we eliminate choices beyond an adjustable nesting level that satisfy one of the following conditions: (1) choices whose alternatives are unifiable, and (2) choices whose alternatives contain errors in the same places. These two conditions ensure that the eliminated choices are unlikely to contribute to type errors. This strategy may fail to eliminate choices, but this happens only when there are already too many type errors in the program, and we therefore stop the inference process and report type errors and change suggestions found so far.

This strategy allows us to maintain choices whose corresponding locations are likely sources of type errors and discard those that are not. Note, however, that this strategy sacrifices the completeness property captured in Theorem 8. We have evaluated the running time and the precision of error diagnosis against the choice nesting levels (see Section 7). We observed that only in very rare cases will the choice nesting level reach 17, a value that variational typing is able to deal with decently (Chen *et al.*, 2014).

Finally, we briefly describe a set of simple heuristics that define the ranking of type and expression changes. Each heuristic assigns a numerical score to each changed location in a message, and we order messages by the sums of the scores from high to low. (1) We prefer places that have deduced expression changes (see Section 6) because these changes reflect common editing mistakes (Lerner *et al.*, 2007). Specifically, we assign messages with an expression change the numerical value 3. Other messages get the value 1. (2) We favor changes that are lower in the abstract syntax trees because changes at those places have a smaller effect on the context and are less likely to introduce exotic results. Roughly speaking, we can partition all functions in a file into layers such that layer 1 contains all the functions

Input: e and Γ

$(\_, \phi, \Delta) \leftarrow infer(\Gamma, e)$
$\mathscr{D} \leftarrow \bigtimes_{\{D \mid (l, D\langle\phi,\phi'\rangle)\in\Delta\}} \{D.1, D.2\}$ – $\mathscr{D}$ represents the whole change space
$i \leftarrow 1$ – Generate messages that change single locations
$\mathscr{M} \leftarrow \varnothing$ – $\mathscr{M}$ collects all the messages that change i locations
while `True`
  for each $\overline{s} \in \mathscr{D}.i \wedge \lfloor\phi\rfloor_{\overline{s}} \neq \bot$ – For each change that removes the type error
    $P \leftarrow \{(l, \lfloor\phi_1\rfloor_{\overline{s}}, \lfloor\phi_2\rfloor_{\overline{s}}) \mid (l, D\langle\phi_1, \phi_2\rangle) \in \Delta \wedge D.2 \in \overline{s}\}$
    $F \leftarrow \{(l, \tau_i, \tau_e, McAdam(\tau_i, \tau_e)) \mid (l, \tau_i, \tau_e) \in P\}$
    $\tau_r \leftarrow \lfloor\phi\rfloor_{\overline{s}}$
    $\mathscr{M} \leftarrow \mathscr{M} \cup \{(F, \tau_r)\}$

  for each $M \in rank(\mathscr{M})$
    present M to the user – The whole process is terminated if M is accepted

  – If no message was accepted
  $i \leftarrow i + 1$ – Generate messages that change one more locations
  $\mathscr{M} \leftarrow \varnothing$

Fig. 10. The error message generation process.

that are not called by any other functions, layer 2 contains all the functions that are called at least once from layer 1, and layer n contains all the functions that are called at least once from layer $n-1$. For example, for the `palin` example from Section 1.1, `palin` belongs to layer 1, `rev` belong to layer 2, and `fold` and `flip` belong to layer 3. We assign n to a message if its involved location is within a function belonging to layer n. Multiple messages pointing to the same function are ordered similarly by regarding `let`, `where`, or anonymous functions to be in lower layers. (3) We prefer changes that have minimal shape difference between the inferred type and the expected type. For example, a change that does not influence the arities of function types is ranked higher than a change that does change arities. Specifically, we assign 3 to messages that do not change arities, 2 to those that change arities by 1 or 2, and 1 to others. Expression changes are considered as not changing arities.

Based on these three heuristics, each message that changes one location (n locations) will be assigned three values ($3 \times n$ values). All the messages that change the same number of locations are ordered by the sum of these numbers, and those with larger values are presented first. In case of ties, we use some additional rules, such as preferring changes of constants over variable references. If messages are still tied, we order them by their reported locations, from left to right in the program text. Messages that change more locations are considered only if all messages changing fewer locations have been rejected by the user.

6 Generating error messages

Given an expression e and the type environment Γ, we generate the error messages following the algorithm sketch shown in Figure 10. The first step is to use *infer*, defined in Section 5, to compute the type potential. $\mathscr{D}$ represents the complete

change space, which is computed incrementally. We use the notation $\mathscr{D}.i$ to get all the decisions that change the expression at i locations. In other words, for each $\bar{s}$ in $\mathscr{D}.i$, there are i unique Ds such that $D.2 \in \bar{s}$.

For each decision $\bar{s}$ in $\mathscr{D}.i$, we compute an error message if $\lfloor\phi\rfloor_{\bar{s}}$ is not $\perp$. Note $\lfloor\phi\rfloor_{\bar{s}}$ is the type of the expression if the changes specified in $\bar{s}$ are applied. Each member of P is a triple (l, τ_i, τ_e), where l is the location of the expression to be changed, τ_i is the inferred type for the expression at l, and τ_e is the expected type to remove the type error. Given τ_i and τ_e, we use an extended version of McAdam's approach (McAdam, 2002a), discussed below, to deduce expression changes, which are stored in F. Using the function *rank*, which implements all the rules descried in Section 5, we then rank all computed messages and present them iteratively to the user. The error debugging process is terminated if a message is accepted by the user. Once we run out of messages for changing i locations, we increment the counter i and generate, rank, and present the next batch of messages for changing $i + 1$ locations.

Instead of presenting each message iteratively, we could envision several ways of integrating CF typing into a user interface. First, in addition to a source code editing panel, we could have an error message panel that can show, say, three messages at a time. This allows the user to quickly compare different error messages. Second, we could add a small numbered mark to each identified error location. Each number indicates how the corresponding location is ranked in comparison to other locations. When the mouse hovers over such a location, we can show more information, such as the inferred type and the expected type of the expression and the type of the whole function if the current expression is changed. We leave the construction of such a user interface for future work.

While it is generally impossible to deduce expression changes from type changes, there are several idiosyncratic situations in which type changes do point to likely expression changes. These situations can be identified by unifying both types of a type change where the unification is performed modulo a set of axioms that represent the pattern inherent in the expression change.

As an example, consider the following expression.[14]

```
zipWith (\(x,y) -> x+y) [1,2] [3,4]
```

Our type change inference suggests to change `zipWith` from its original type `(a -> b -> c) -> [a] -> [b] -> [c]` to something of type `((Int,Int) -> Int) -> [Int] -> [Int] -> d`. Given these two types, we can deduce to curry the first argument to the function `zipWith` to remove the type error. (At the same time, we substitute `d` in the result type with `Int`.)

By employing unification modulo different theories, McAdam (McAdam, 2002a) has developed a theory and an algorithm to systematically deduce changes of this sort. We have adopted this approach (and extended it slightly) for deducing expression changes, such as swapping the arguments of function calls, currying,

[14] This example is adapted from Lerner *et al.* (2007), where `zipWith` is called `map2`.

and uncurrying of functions, or adding and removing arguments of function calls.

The extension is based on a simple form of identifying non-arity-preserving type changes. If such a change is used to modify the types, then McAdam's approach is applied, and the result is then interpreted in light of the non-arity-preserving type change as a new form of expression change. As an example, here is the method of identifying the addition or removal of arguments to function calls. In this case, the differences in the two types to be unified will lead to a second-level type change that pads one of the types with an extra type variable. For example, given the inferred type $\tau_1 \to \tau_3$ and the expected type $\tau_1 \to \tau_2 \to \tau_3$, we turn the first type into $\alpha \to \tau_1 \to \tau_3$. The application of McAdam's approach suggests to swap the arguments. Also, α is mapped to τ_2. Interpreting the swapping suggestion through the second-level type change of padding, we deduce the removal of the second argument.

Besides these systematic change deductions, we also support some *ad hoc* expression changes. Specifically, we infer changes by inspecting the expected type only. For example, if the inferred type for `f` in `f g e` is `b -> c` while the expected type is `(a -> b) -> a -> c`, we suggest to change `f g e` to `f (g e)`. Another example is given by situations in which the result type of an expected type matches exactly one of its (several) argument types. In that case, we suggest to replace the whole expression with the corresponding argument. This case applies, in fact, to the `palin` example, where the type change for `(:)` is to replace `a -> [a] -> [a]` by `a -> [b] -> a`. We therefore infer to replace `(:) z []`, which is `[z]`, by `z` because the first argument type is the same as the return type. Another case is when in expression `f g h` the expected type for `f` is `(a -> b) -> a -> b`. Then, we suggest to remove `f` from the expression. There are several other similar *ad hoc* changes that are useful in some situations, but we do not discuss them here.

In Section 8, we compare our method with McAdam's original approach. Here, we only note that the success of the method in our prototype depends to a large degree on the additional information provided by type-change inference, specifically, the more precise and less biased expected types that are used for the unification.

7 Evaluation

To evaluate the usefulness and efficiency of the CF typing approach, we have implemented a prototype of type-change inference and expression-change deduction in Haskell. (In addition to the constructs shown in Section 4, the prototype also supports some minor, straightforward extensions, such as data types and case expressions.) We compare the results produced by our CF typing tool to Seminal (Lerner *et al.*, 2006, 2007), Helium (Heeren *et al.*, 2003; Heeren, 2005), and GHC. There are several reasons for selecting this group of tools. First, they provide currently running implementations. Second, these tools provide a similar functionality as CF typing, namely, locating type errors and presenting change suggestions, both at the type and the expression level. We have deliberately excluded

slicing tools from the comparison because they only show all possible locations, and do not suggest changes.[15]

In Section 7.1, we report the evaluation results for a set of programs that were gathered from a wide range of publications on type error debugging.[16] One of the reasons for Seminal's comparatively modest performance is due to the fact that it systematically deals with multiple errors in programs, whereas the examples in Section 7.1 focus on single errors. We therefore specifically investigate the performance of CF typing in the presence of multiple errors in Section 7.2. After that, we investigate the running time aspect of CF typing in Section 7.3.

7.1 Evaluation of programs from the literature

For evaluating the applicability and accuracy of the tools, we have gathered a collection of 121 examples from 22 publications about type-error diagnosis. These papers include recent Ph.D. theses (Yang, 2001; McAdam, 2002a; Heeren, 2005; Wazny, 2006) and papers that represent most recent and older work (Schilling, 2012; Lerner *et al*., 2007; Johnson & Walz, 1986). These papers cover many different perspectives of the type-error debugging problem, including error slicing, explanation systems, reordering of unification, automatic repairing, and interactive debugging. Since the examples presented in each paper have been carefully chosen or designed to illustrate important problem cases for type-error debugging, we have included them all, except for examples that involve type classes since our tool (as well as Seminal) does not currently support type classes. This exclusion did not have a significant effect. We gathered eight unique examples regarding type classes involved in type errors discussed in Stuckey *et al*. (2003) and Wazny (2006). Both GHC and Helium were able to produce a helpful error message in only 1 case. Otherwise, the examples range from very simple, such as `test = map [1,10] even` to very complex ones, such as the `plot` example introduced in Wazny (2006). The program sizes range from 1 LOC to 15 LOC.

We have grouped the examples into two categories. The first group ("with Oracle") contains 86 examples for which the correct version is known (because it either is mentioned in the paper or is obvious from the context). The other group ("ambiguous") contains the remaining 35 examples that can be reasonably fixed by several different single-location changes. For the examples in the "with Oracle" group, we have recorded how many correct suggestions each tool can find with at most n attempts. For the examples in the "ambiguous" group, we have determined how often a tool produces a complete, partial, or incorrect set of suggestions. For example, for the expression `\f g a -> (f a, f 1, g a, g True)`, which is given in Bernstein & Stark (1995), Helium suggests to change `True` to something of

[15] There are a few interactive approaches that have been proposed (Chitil, 2001; Stuckey *et al*., 2003), but they currently do not provide running implementations. Moreover, Chameleon (Wazny, 2006) has evolved to focus on typing extensions of the Haskell type system. Since the tool has switched off its type-debugging facilities, it is not a viable candidate for comparison.

[16] Available at `http://www.ucs.louisiana.edu/ sxc2311/ws/files/cft-bench.hs`

	86 examples with Oracle					35 ambiguous examples		
	1	2	3	≥ 4	never	complete	partial	incorrect
CF typing	67.4	80.2	88.4	91.9	8.1	100.0	0.0	0.0
Seminal	47.7	54.7	58.1	59.3	40.7	40.0	25.7	34.3
Helium	61.6	-	-	61.6	38.4	0.0	100.0	0.0
GHC	17.4	-	-	17.4	82.6	0.0	34.3	65.7

Fig. 11. Evaluation results for different approaches over 121 collected examples (in %). The column headers are the numbers of messages. For example, the entry with the row header "CF typing" and the column header "3" indicates that with up to 3 messages, CF typing can locate 88.4% of all the type errors in the 86 examples with oracle.

type `Int`. While this is correct, there are also other changes possible, for example, changing `f 1` to `f True`. Since these are not mentioned, the result is categorized as partial.

Figure 11 presents the results for the different tools and examples with unconstrained choice nesting level for CF typing. Note that GHC's output is considered correct only when it points to the correct location and produces an error message that is not simply reporting a unification failure or some other compiler-centric point of view. We have included GHC only as a baseline since it is widely known. The comparison of effectiveness is meant to be between CF typing, Seminal, and Helium.

The numbers show that CF typing performs overall best. Even if we only consider the first change suggestion, it outperforms Helium that comes in second. Taking into account second and third suggestions, Seminal catches up, but CF typing performs even better.

In cases where Helium produces multiple suggestions, all suggestions are wrong. For CF typing 21 out of the 58 correct suggestions (that is, 36%) are expression changes. For Seminal the numbers are 20 out of 41 (or 51%), and for Helium it is 15 out of 52 (or 29%). This shows that Seminal produces a higher rate of expression change suggestions at a lower overall correctness rate.

Most of Helium and Seminal's failures are due to incorrectly identified change locations. Another main reason for Seminal's incorrect suggestions is that it introduces too extreme changes.

Most cases for which CF typing fails are caused by missing parentheses. For example, for the expression `print "a" ++ "b"` (Lerner *et al.*, 2007), our approach suggests to change `print` from the inferred type `a -> IO ()` to the type `String -> String` or change `(++)` from the expected type `[a] -> [a] -> [a]` to the inferred type `IO () -> String -> String`. Neither of the suggestions allows us to deduce the regrouping of the expression.

To summarize, since the examples that we used have been designed to test very specific cases, the numbers do not tell much about how the systems would perform in everyday practice. They provide more like a stress test for the tools, but the direct comparison shows that CF typing performs very well compared with other tools and thus presents a viable alternative to type debugging.

```
The definition of rev is
the same as before and is omitted

palin xs = rev xs == xs

res xs = palin xs && not 1
```

```
fold f z [] = [z]
fold f z (x:xs) = fold f (f z x) xs
flip f x y = f y x
rev = fold (flip (:)) False

palin xs = rev xs == xs
```

Fig. 12. Type errors that are independent (left) and entangled (right). The gray background marks the second type error in each program.

	30 examples with 2 independent errors				30 examples with 2 entangled errors			
	1	2	3	≥ 4	1	2	3	≥ 4
CF typing	66.6	80.0	86.7	93.3	0.0	0.0	13.3	46.7
Seminal	43.3	50.0	60.0	66.6	0.0	0.0	11.7	33.3

Fig. 13. Accuracy of error reporting for CF typing and Seminal for programs with multiple errors (in %). The column headers have the same meaning as those in Figure 11.

7.2 Evaluation of programs containing multiple errors

Since most ill-typed programs reported in the type error debugging literature contain only single type errors, we have manually created programs containing multiple type errors for the evaluation. We first focus on the case that each program has two type errors. We say that two type errors are *independent* if the presence of one does not have any impact or causes a negative impact on the debugging of the other. Dually, we call two type errors that are not independent *entangled*. Consider the examples in Figure 12.

On the left of Figure 12, both `palin` and `not 1` contain a type error. Those two errors are independent, and to make the program well typed, we have to make two changes. Two independent type errors may be close (like the ones shown) or far apart. On the right of Figure 12, the use of `False` (which should be `[]`) in `rev` adds a second type error that is entangled with the first type error in `fold` because it prevents or postpones the identification of the first one. The reason for this effect of entangled type errors is that there are many single-location updates that will fix both type errors, and this causes typing approaches that favor the identification of single-location updates, such as CF typing, to be misled by the second type error.

For example, CF typing will first suggest to change `f` in `f z x` from type `[a] -> a -> [a]` to `Bool -> Bool -> Bool`, which will fix both type errors. The second suggestion generated by CF typing is to change `(:)` from type `a -> [a] -> [a]` to `Bool -> Bool -> Bool`, which will also fix both type errors. As a result, the fixes regarding `[z]` and `False` will be presented later. Seminal reports for this program the type error at `fold` in `rev` in its first message and `(:)` in `flip (:)` in its second message, missing the real error causes in both messages.

To evaluate how CF typing and Seminal behave on programs with multiple errors, we have created 30 programs with independent errors by randomly merging the programs from the set we collected from the literature (Section 7.1). We have also created 30 programs with entangled errors by modifying the collected programs.

Figure 13 presents the evaluation result for those programs. For programs with independent errors, each message generated by CF typing contains change suggestions for two locations because both of them have to be changed to make

the program well typed. For entangled errors, CF typing may generate messages that contain only one change suggestion because changing a single location could already fix both type errors, as discussed earlier. Also, each error message may fix one type error but miss the other one. For each message, we assign a value 0%, 50%, or 100% if the message does not fix any error, fixes one error, or fixes both errors, respectively. Each entry in Figure 13 is the average of the values over all 30 programs.

We can observe that for independent errors, the result of CF typing and Seminal is similar to that in Figure 11. This indicates that the presence of multiple independent errors does not cause problems for type error debugging approaches that favor single-location updates. However, entangled errors are more problematic. As Figure 11 shows, the precision of both CF typing and Seminal decreases significantly. In particular, with the first two messages, neither CF typing nor Seminal could locate any type error correctly. The main reason, as we have seen already, is that both CF typing and Seminal prefer to change as few locations as possible, which may lead to the identification of less likely causes. (Note that for this analysis we considered for each program only up to eight messages, since it is not very likely that the user will go through more messages to fix the type error.)

Besides the programs with two type errors, we have also created 30 programs with four type errors by randomly merging the 60 programs we created with two type errors. These programs now contain a mix of independent and entangled errors. We have evaluated them using the same evaluation scheme as for programs with two type errors, that is we assign a value of 0% to 100% with a 25% interval for each message which fixes 0–4 type errors, respectively.

The precision for CF typing with 1, 2, 3, and $\geqslant 4$ messages are 32.5%, 38.4%, 44.2%, and 65.4%, respectively. The result for Seminal with 1, 2, 3, and $\geqslant 4$ messages are 21.4%, 24.7%, 30.6%, and 39.2%, respectively. (Note that, as before, we investigated up to eight messages for each program.)

Overall, our conclusion regarding the debugging of programs with multiple type errors is that CF typing performs better than Seminal, and both of them work poorly for entangled errors. In fact, most existing debuggers prefer to change as few locations as possible and will therefore probably not work well for entangled errors. An interesting question for future work is to investigate how often entangled errors happen in practice.

7.3 Performance

With the help of variational typing, we can generate all the potential changes very efficiently. The running time for all the collected examples (Section 7.1) is within 2 seconds. Figure 14 shows the running time for both our approach and Seminal for processing the reported examples. For each point (x, y) on the curve, it means that x% of all examples are processed with y seconds. The running time for our approach is measured on a laptop with a 2.8GHz dual core processor and 3GB RAM running Windows XP and GHC 7.0.2. The running time for Seminal is measured on the

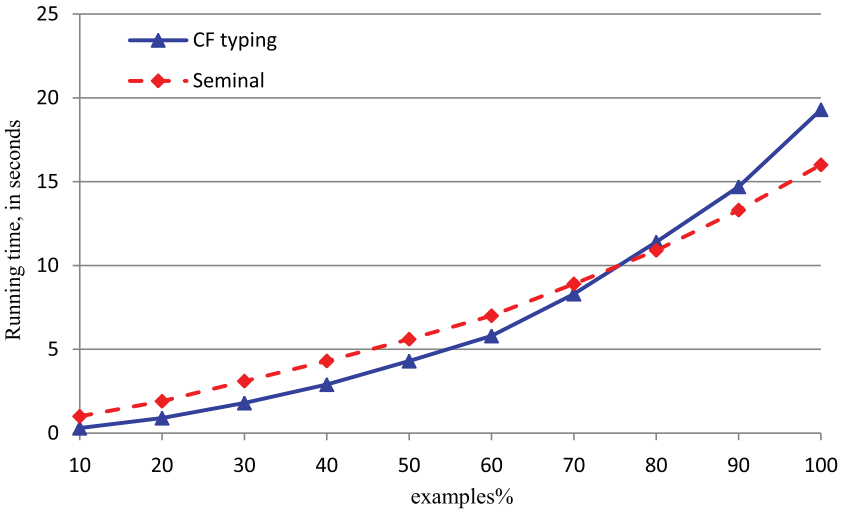

Fig. 14. Running time for typing x% of the examples. The time reported is the accumulated time for 10 runs.

same machine with Cygwin 5.1. The purpose of the graph is simply to demonstrate the feasibility of our approach.

To gauge the impact of the nesting level restriction, we have also measured the running time of CF typing on a number of programs from the Helium benchmarks (Hage, 2013), the program sizes of which range from 2 LOC to 190 LOC. Specifically, we randomly picked 200 programs: 50 each of the size 50 LOC, 100 LOC, 150 LOC, and 190 LOC. The following table gives a summary of the running time (in seconds) of CF typing on these programs:

	Maximum nesting level: None			Maximum nesting level: 17		
	Min	Max	Average	Min	Max	Average
50 LOC	0.3	11.4	5.6	0.3	2.1	1.6
100 LOC	0.7	23.4	12.6	0.7	3.8	2.5
150 LOC	1.3	89.7	47.4	1.3	5.1	3.8
190 LOC	2.3	374.5	147.9	2.3	7.5	5.4

We observe that the running time varies quite a bit. The reason is that different programs have very different dependencies and used very different functions. The programs on which CF typing runs fastest (column min in the table) contain small independent functions, whereas the ones for which CF typing needs the most time (column max in the table) contain functions that have a linear dependency, that is one function is dependent on another, which is in turn dependent on a third function, and so on. The average time is a division of the total running time for programs of a specific size over 50. We can observe that CF typing slows down quite quickly as program size increases when we do not set a maximum choice nesting level. The reason is that it computes all potential type error fixes at once. As the above table shows, the performance issue can mostly be mitigated by setting a maximum choice nesting level, such as 17. Overall, CF typing is efficient enough for

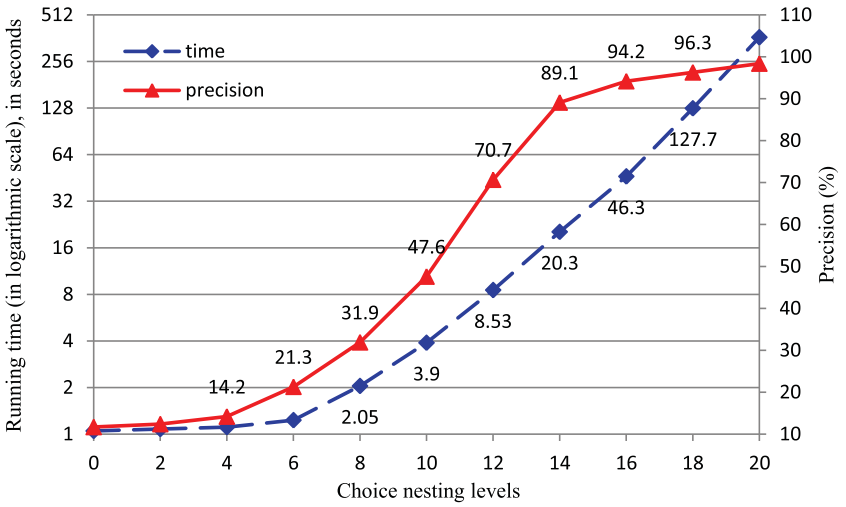

Fig. 15. Limits on choice nesting trade efficiency for precision.

most student programs from the benchmark. Also, as pointed out by Lerner et al. (2007), the short time spent by a tool often pays off, since students usually spend a long time on debugging type errors.

While the idea of setting a maximum choice nesting level can address the performance issue with CF typing, it is unclear about its impact on error localization precision of CF typing. For this purpose, we have automatically generated large examples, and we use functions of types such as $\alpha \to \alpha \to \ldots \to \alpha$ to trigger the choice elimination strategies discussed in Section 5. We first generated 200 type correct examples and then introduced one or two type errors in each example by changing the leaves, swapping arguments, and so on. Each example contains about 60,000 nodes in its tree representation.

Figure 15 presents the running time and precision against choice nesting levels for these generated examples. A change suggestion is considered correct if it fixes a type error and appears among the first four changes for that example. Precision is measured by the quotient of the number of examples that have correct change suggestions and the number of all examples. From the figure, we observe that a nesting level cut-off between 12 and 18 achieves both high precision and efficiency.

8 Related work

We have grouped our discussion of related work according to major features shared by the different approaches.

Reporting single locations: Most of the single-error-location approaches are based on some variant of the algorithm $\mathscr{W}$ and report an error as soon as the algorithm fails. Since the original algorithm $\mathscr{W}$ is biased in the order in which unification problems are solved (which has a negative impact on locating errors), many approaches have tried to eliminate this bias. Examples are algorithms $\mathscr{M}$ (Lee & Yi, 2000), $\mathscr{G}$ (Eo

et al., 2004), $\mathcal{W}^{SYM}$ and $\mathcal{M}^{SYM}$ (McAdam, 2002b), and $\mathcal{U}_{\mathbf{AE}}$ and $I\&I$ (Yang *et al*., 2000). All these algorithms interpret the place of unification failure as the source of the type error. In contrast, Johnson and Walz (1986) and the Helium tool (Heeren *et al*., 2003; Heeren, 2005) use heuristics to select the most likely error location from a set of potential places. Although heuristics often work well and lead to more accurate locations, they can still get confused due to the single-location constraint. In contrast, we explore all potential changes and rank them from most to least likely. Recently, Seidel *et al*. (2017) developed a machine learning-based approach for locating most likely error locations.

In deducing expression changes from type changes, we have used (an extension of) McAdam's technique (2002a). Since his approach is based on the algorithm $\mathcal{W}$, it suffers from the bias of error locating mentioned above. Moreover, his approach does not have access to the precise expected type, which helps in our approach to ensure that deduced expression changes will not have an impact on the program as a whole.

Explaining type conflicts: Some approaches have focused on identifying and explaining the causes of type conflicts. Wand (1986) records each unification step so that they can be tracked back to the failure point. Duggan and Bent (1995) on the other hand record the reason for each unification that is being performed. Beaven and Stansifer (1994) and Yang (2000) produce textual explanation for the cause of the type errors.

While these techniques can be useful in many cases, there are also potential downsides. First, the explanation can become quite verbose and repetitive, and the size grows rapidly as the program size increases. Second, the explanation is inherently coupled to the underlying algorithm that performs the inference. Thus, knowledge about how the algorithm works is often needed to understand the produced messages. Third, the explanations usually lead to the failure point, which is often the result of biased unification and not the true cause of the type error. Finally, although a potential fix for the type error may lurk in the middle of the explanation chain, it is not always clear about how to exploit it and change the program.

Recently, Seidel et al. (2016) developed an explanation approach from a very different perspective. Given an ill-typed expression, their approach finds an input such that the evaluation of the expression will fail with the given input.

Interactive debugging: While many tools attempt to improve the static presentation of type error information, interactive approaches give users a better understanding about the type error or why certain types have been inferred for certain expressions. Consequently, several approaches to interactive type debugging have been pursued.

The ability to infer types for unbound variables enables a type debugging paradigm that is based on the idea of replacing a suspicious program snippet by a fresh variable (Bernstein & Stark, 1995). If such a replacement leads to a type-correct program, then the error location has been identified. However, the original system proposed by Berstein and Stark requires users to perform these steps manually. Later, Braßel (2004) automated this process by systematically commenting out parts of the

program and running the type checker iteratively. This method also suffers from the right bias problem unless all expression leaves are considered for replacement. Moreover, it is unclear how to handle programs that contain more than one type error.

Through employing a number of different techniques, Chitil (2001), Neubauer and Thiemann (2003), Wazny (2006), and Stuckey *et al.* (2003) have developed tools that allow users to explore a program and inspect the types for any subexpression. Chameleon (Stuckey *et al.*, 2003; Wazny, 2006) also allows users to query how the types for specific expressions are inferred. All these approaches provide a mechanism for users to explore a program and view the type information. However, none of them provides direct support for finding or fixing type errors.

Building on the idea from this paper, we have developed a method called *guided type debugging* (Chen & Erwig, 2014b), which allows users to specify an expected type for an ill-typed expression. The guided type debugger will then find updates that transform the ill-typed expression to have the user-specified type.

Error slicing: The main advantage of slicing approaches (Tip & Dinesh, 2001; Haack & Wells, 2003; Schilling, 2012) is that they return all locations related to type errors. The downside is that they cover too many locations. Recent improvements in Chameleon (Wazny, 2006) have helped to reduce the number of locations, but the problem still persists (recall the example in the Introduction). Moreover, slicing tools do not provide suggestions for how to get rid of the type error. A problem with slicing approaches and their improvements is that it is quite difficult to achieve a good balance to report fewer locations while not to miss the real error cause. This problem was recently addressed by Pavlinovic *et al.* (2014; 2015), who developed a method that can find comprehensive error causes but present them iteratively using some ranking criterion. Our CF typing approach also implements this idea but additionally provides informative error messages while their method only locates error causes.

Probabilistic approaches: SHErrLoc (Zhang & Myers, 2014; Zhang *et al.*, 2015) debugs type errors in the following steps. First, it transforms type constraints generated by compilers into a graph representation, where nodes are types and edges are relations between types. For example, two types that are required to be the same are connected with an edge. Also, if a type is a component of another type, for example the argument type of a function type, then they are also connected. After that, each edge is classified as satisfiable or unsatisfiable. Unsatisfiable edges cause type errors. Finally, a Bayesian model is applied to determine which node (and the constraints that correspond to incident edges of that node) most likely causes the type error. Intuitively, SHErrLoc selects the node that is shared by more unsatisfiable edges but does not belong to satisfiable edges as the error cause. SHErrLoc can also generate error slices by reporting all nodes that are involved in unsatisfiable edges. Besides the technical differences between SHErrLoc and our

approach, the error messages generated by our approach include more information, as they contain at least the expected type and the inferred type of the error cause and also the resulting type if the suggested change is followed. In contrast, SHErrLoc reports type errors in constraints only, as can be seen from Section 1.1.

Embracing type uncertainty: Instead of choice types, one can encode variational types also using sum types. This is the approach taken by Neubauer and Thiemann (2003), who developed a type system based on discriminative sum types to record the causes of type errors. Specifically, they place two non-unifiable types into a sum type. Named choice types as employed by CF typing provide more fine-grained control over variations in types than discriminative sum types. While sum types are unified component-wise, this is only the case for choice types of the same name. Each alternative in a choice type is unified with all the alternatives in other choices with different names. Also, their system returns a set of sources related to type errors. Thus, it can be viewed as an error slicing approach. However, compared to other slicing approaches, it is not guaranteed that the returned set of locations is minimal. Moreover, the approach does not provide specific change locations or change suggestions.

Typing by searching: CF typing and Seminal (Lerner *et al*., 2006; Lerner *et al*., 2007) could both be called "search based," although the search happens at different levels. While CF typing explores changes on the type level, Seminal works on the expression level directly. Given an ill-typed program, Seminal first has to decide where the type error is. Seminal uses a binary search to locate the erroneous place. Once the problematic expression is found, Seminal searches for a type-corrected program by creating mutations of the original program, for example, by swapping the arguments to functions, currying or uncurrying function calls, and so on. This way of searching causes Seminal to make mistakes in locating errors when the first part of the program itself does not contain a type error but actually triggers type errors because it is too constrained. For example, the cause of the type error in the `palin` example discussed in Section 1 is the `fold` function, which is itself well typed. As a result, Seminal fails to find a correct suggestion. Our approach does not suffer from this problem since the generation of type changes equally varies all AST leaves in the program.

Tsushima and Chitil (2014) also developed a search method by using an exiting type checker. Given an ill-typed expression e, they transformed it into λ`hole`.e[`hole`], where e[`hole`] replaces one leaf of e with a `hole`. Their approach then calls an existing type checker to check the transformed expression. If the new expression type checks, then an error location has been identified. Moreover, the parameter type of the new expression is the expected type of the leaf and the return type of the new expression is the type of e if the leaf has the suggested type. Compared to our approach, their approach does not reuse computations when considering changing different leaves. Moreover, their approach does not work when the expression has more than one type error.

9 Conclusions

We have presented a new method for debugging type errors. The approach is based on the notion of CF typing, which is the idea of systematically varying the types of all AST leaves to generate a typing potential for the erroneous program that can be explored and reasoned about. We have exploited this typing potential and the associated set of type changes to create a ranked list of type-change and expression-change suggestions that can eliminate type errors from programs. A comparison of a prototype implementation with other tools has demonstrated that the approach works very well and, in fact, outperforms its competitors.

In future work, we plan to investigate other uses of typing potentials and type changes. For example, we may extract more information from typing potentials about why certain error messages have been generated. Specifically, besides showing the expected type in each error message, we may show the related context that coerces the erroneous expression to have the expected type. Finally, we plan to investigate how well the approach works for the debugging of type errors in richer type systems.

References

Beaven, M. & Stansifer, R. (1994) Explaining type errors in polymorphic languages. *ACM Lett. Program. Lang. Syst.* **2**, 17–30.

Bernstein, K. L. & Stark, E. W. (1995) *Debugging Type Errors*. Tech. rept. State University of New York at Stony Brook.

Braßel, B. (2004) Typehope: There is hope for your type errors. In Proceedings of International Workshop on Implementation of Functional Languages.

Chambers, C., Chen, S., Le, D. & Scaffidi, C. (2012) The function, and dysfunction, of information sources in learning functional programming. *J. Comput. Sci. Colleges.* **28**(1), 220–226.

Chen, S. & Erwig, M. (2014a) Counter-factual typing for debugging type errors. In Proceedings of ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 583–594.

Chen, S. & Erwig, M. (2014b) Guided type debugging. In Proceedings of International Symposium on Functional and Logic Programming, LNCS, vol. 8475, pp. 35–51.

Chen, S., Erwig, M. & Walkingshaw, E. (2012) An error-tolerant type system for variational lambda calculus. In Proceedings of ACM International Conference on Functional Programming, pp. 29–40.

Chen, S., Erwig, M. & Walkingshaw, E. (2014) Extending type inference to variational programs. *ACM Trans. Program. Lang. Syst.* **36**(1), 1:1–1:54.

Chen, S., Erwig, M. & Smeltzer, K. (2017) Exploiting diversity in type checkers for better error messages. *J. Vis. Lang. Comput.* **39**(C), 10–21.

Chitil, O. (2001 September) Compositional explanation of types and algorithmic debugging of type errors. In Proceedings of ACM International Conference on Functional Programming, pp. 193–204.

Choppella, V. (2002) *Unification Source-Tracking with Application to Diagnosis of Type Inference*. Ph.D. thesis, Indiana University.

Damas, L. & Milner, R. (1982) Principal type-schemes for functional programs. In Proceedings of ACM Symposium on Principles of Programming Languages, pp. 207–212.

Duggan, D. & Bent, F. (1995) Explaining type inference. *Sci. Comput. Program.* **27**(1), 37–83.

Eo, H., Lee, O. & Yi, K. (2004) Proofs of a set of hybrid let-polymorphic type inference algorithms. *New Gener. Comput.* **22**(1), 1–36.

Erwig, M. (2006) Visual Type Inference. *J. Vis. Lang. Comput.* **17**(2), 161–186.

Erwig, M. & Walkingshaw, E. (2011) The choice calculus: A representation for software variation. *ACM Trans. Softw. Eng. Methodol.* **21**(1), 6:1–6:27.

Haack, C. & Wells, J. B. (2003) Type error slicing in implicitly typed higher-order languages. In Proceedings of European Symposium on Programming, pp. 284–301.

Hage, J. (2013) Helium Benchmark Programs (2002–2005). Private communication.

Hage, J. & Heeren, B. (2007) Heuristics for type error discovery and recovery. In Proceedings of Implementation and Application of Functional Languages, pp. 199–216.

Heeren, B., Leijen, D. & van I. A. (2003) Helium, for learning haskell. In Proceedings of ACM SIGPLAN Workshop on Haskell, pp. 62–71.

Heeren, B. J. (2005) *Top Quality Type Error Messages*. Ph.D. thesis, Universiteit Utrecht, The Netherlands.

Johnson, G. F. & Walz, J. A. (1986) A maximum-flow approach to anomaly isolation in unification-based incremental type inference. In Proceedings of ACM Symposium on Principles of Programming Languages, pp. 44–57.

Lee, O. & Yi, K. (1998) Proofs about a folklore let-polymorphic type inference algorithm. *ACM Trans. Program. Lang. Syst.* **20**(4), 707–723.

Lee, O. & Yi, K. (2000) *A Generalized Let-Polymorphic Type Inference Algorithm*. Tech. rept. Technical Memorandum ROPAS-2000-5, Research on Program Analysis System, Korea Advanced Institute of Science and Technology.

Lerner, B., Flower, M., Grossman, D. & Chambers, C. (2007) Searching for type-error messages. In Proceedings of ACM International Conference on Programming Language Design and Implementation, pp. 425–434.

Lerner, B., Grossman, D. & Chambers, C. (2006) Seminal: Searching for ml type-error messages. In Proceedings of Workshop on ml, pp. 63–73.

Loncaric, C., Chandra, S., Schlesinger, C. & Sridharan, M. (2016). A practical framework for type inference error explanation. In Proceedings of the 2016 Acm Sigplan International Conference on Object-Oriented Programming, Systems, Languages, and Applications. OOPSLA 2016. New York, NY, USA: ACM, pp. 781–799.

McAdam, B. J. (1999). *Graphs for Recording Type Information*. Technical Report ECS-LFCS-99-415. University of Edinburgh.

McAdam, B. J. (2002a). *Reporting Type Errors in Functional Programs*. Ph.D. thesis, Larboratory for Foundations of Computer Science, The University of Edinburgh.

McAdam, B. J. (2002b). *Repairing Type Errors in Functional Programs*. Ph.D. thesis, University of Edinburgh. College of Science and Engineering. School of Informatics.

Neubauer, M. & Thiemann, P. (2003) Discriminative sum types locate the source of type errors. In Proceedings of ACM International Conference on Functional Programming, pp. 15–26

Pavlinovic, Z., King, T. & Wies, T. (2014) Finding minimum type error sources. In Proceedings of ACM International Conference on Object Oriented Programming Systems Languages & Applications, pp. 525–542

Pavlinovic, Z., King, T. & Wies, T. (2015) Practical SMT-based type error localization. In Proceedings of ACM International Conference on Functional Programming, pp. 412–423.

Pfenning, F. (1991) Unification and anti-unification in the calculus of constructions. In Proceedings of IEEE Symposium on Logic in Computer Science, pp. 74–85.

Schilling, T. (2012) Constraint-free type error slicing. In Proceedings of Trends in Functional Programming. Springer, pp. 1–16.

Seidel, E. L., Jhala, R. & Weimer, W. (2016) Dynamic witnesses for static type errors (or, ill-typed programs usually go wrong). In Proceedings of ACM International Conference on Functional Programming, pp. 228–242.

Seidel, E. L., Sibghat, H., Chaudhuri, K., Weimer, W. & Jhala, R. (2017) Learning to blame: Localizing novice type errors with data-driven diagnosis. *Proc. ACM Program. Lang.* **1**(OOPSLA), 60:1–60:27.

Stuckey, P. J., Sulzmann, M. & Wazny, J. (2003) Interactive type debugging in haskell. InProceedings of ACM Sigplan Workshop on Haskell, pp. 72–83.

Tip, F. & Dinesh, T. B. (2001) A slicing-based approach for locating type errors. *ACM Trans. Softw. Eng. Methodol.*, **10**(1), 5–55.

Tirronen, V., Uusi-mäkelä, S. & Isomöttönen, V. (2015) Understanding beginners' mistakes with Haskell. *J. Funct. Program.*, **25**, 1–31.

Tsushima, K. & Chitil, O. (2014) Enumerating counter-factual type error messages with an existing type checker. In Proceedings of 16th Workshop on Programming and Programming Languages.

Wand, M. (1986) Finding the source of type errors. In Proceedings of ACM Symposium on Principles of Programming Languages, pp. 38–43.

Wazny, J. R. (2006 January) *Type Inference and Type Error Diagnosis for Hindley/Milner with Extensions.* Ph.D. thesis, The University of Melbourne.

Wu, B. & Chen, S. (2017) How type errors were fixed and what students did? *Proc. ACM Program. Lang.* **1**(OOPSLA), 105:1–105:27.

Wu, B., Campora III, John P. & Chen, S. (2017) Learning user friendly type-error messages. *Proc. ACM Program. Lang.* **1**(OOPSLA), 106:1–106:29.

Yang, J. (2000) Explaining type errors by finding the source of a type conflict. In Proceedings of Trends in Functional Programming. Intellect Books, pp. 58–66.

Yang, J. (2001 May). *Improving Polymorphic Type Explanations.* Ph.D. thesis, Heriot-Watt University.

Yang, J., Michaelson, G., Trinder, P. & Wells, J. B. (2000) Improved type error reporting. In Proceedings of International Workshop on Implementation of Functional Languages, pp. 71–86.

Zhang, D. & Myers, A. C. (2014) Toward general diagnosis of static errors. In Proceedings of ACM Symposium on Principles of Programming Languages, pp. 569–581.

Zhang, D., Myers, A. C., Vytiniotis, D. & Peyton-Jones, S. (2015) Diagnosing type errors with class. In Proceedings of ACM Conference on Programming Language Design and Implementation, pp. 12–21.