Counter-Factual Typing for Debugging Type Errors [*]

Sheng Chen
School of EECS, Oregon State University
chensh@eecs.oregonstate.edu

Martin Erwig
School of EECS, Oregon State University
erwig@eecs.oregonstate.edu

Abstract

Changing a program in response to a type error plays an important part in modern software development. However, the generation of good type error messages remains a problem for highly expressive type systems. Existing approaches often suffer from a lack of precision in locating errors and proposing remedies. Specifically, they either fail to locate the source of the type error consistently, or they report too many potential error locations. Moreover, the change suggestions offered are often incorrect. This makes the debugging process tedious and ineffective.

We present an approach to the problem of type debugging that is based on generating and filtering a comprehensive set of type-change suggestions. Specifically, we generate *all* (program-structure-preserving) type changes that can possibly fix the type error. These suggestions will be ranked and presented to the programmer in an iterative fashion. In some cases we also produce suggestions to change the program. In most situations, this strategy delivers the correct change suggestions quickly, and at the same time never misses any rare suggestions. The computation of the potentially huge set of type-change suggestions is efficient since it is based on a variational type inference algorithm that type checks a program with variations only once, efficiently reusing type information for shared parts.

We have evaluated our method and compared it with previous approaches. Based on a large set of examples drawn from the literature, we have found that our method outperforms other approaches and provides a viable alternative.

Categories and Subject Descriptors F.3.3 [*Logics and Meanings of Programs*]: Studies of Program Constructs–Functional constructs,Type structure; D.3.2 [*Programming Languages*]: Language Classifications–Applicative (functional) languages; D.2.5 [*Software Engineering*]: Testing and Debugging

General Terms Languages, Theory

Keywords Type inference, error localization, type error messages, choice types, change suggestions, type-error debugging

[*] This work is supported by the National Science Foundation under the grants CCF-0917092, CCF-1219165, and IIS-1314384.

POPL'14 January 22 - 24 2014, San Diego, CA, USA
ACM 978-1-4503-2544-8/14/01...$15.00. Copyright is held by the owner/author(s). Publication rights licensed to ACM.
http://dx.doi.org/10.1145/2535838.2535863.

1. Introduction

Generating informative and helpful type error messages remains a challenge for implementing type inference algorithms. Soon after the algorithm $\mathcal{W}$ [8] had been developed, this problem was recognized [15, 26]. It has since prompted numerous research efforts [7, 12, 14, 16–18, 20, 21, 23, 25, 27, 29]. Although considerable progress has been made, 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 [24]. 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) 7.6[1] produces the following error message.[2]

```
Occurs check: cannot construct the infinite type: t0 = [t0]
Expected type: [[t0]]
  Actual type: [t0]
In the second argument of '(==)', namely 'xs'
In the expression: rev xs == xs
```

While technically accurate, the error message doesn't directly point to the source of the error, and it doesn't tell the user how it could be fixed either. The use of compiler jargon makes the error message 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 some effort to manually reconstruct some of the types and solve unification problems. (Hugs98[3] produces a similar error message and suffers from the same problems.)

One of the problems of the approach taken in GHC is that it commits to a *single* error location, because in some cases the program text does not contain enough information to confidently make the right decision about the correct error location. This has led to a number of program slicing approaches that try to identify a *set* of possible error locations instead.

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[4] type error slicer for SML [12] produces the following

[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] `www.haskell.org/hugs/`

[4] `www.macs.hw.ac.uk/ultra/skalpel/`

result. (We have translated the program into ML for Skalpel to work.)

```
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  ;
```

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,[5] 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 solving and identifies a minimal set of unsatisfiable constraints, from which the corresponding places in the program contributing to the type error are derived. While Chameleon 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.

On the other end of the single-vs.-many location spectrum we find approaches that, like GHC or Hugs, follow Johnson and Walz's idea of finding the most likely erroneous location and try to add explanations or suggestions for how to correct the error. One example is the Helium compiler[6], which was developed to support the teaching of typed functional programming languages. A declared focus of Helium is to generate good error messages [14]. 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 doesn't perform much better than GHC or Hugs on this example and provides feedback only in terms of internal representations used by the compiler.

Seminal[7] is a tool for type checking ML programs that also produces change suggestions [18, 19]. Seminal blames the type error on the function `palin` and suggests the following corrective change.

```
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)) ;;
```

Unfortunately, the suggested error location is not correct (according to [24]), and 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`.)

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 doesn't work. In the shown example, both change-suggesting tools fail to correctly locate the error location.

The tasks 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 expression `not 1`. The error in this expression is either `not` or `1`,[8] 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 fact provides a strong justification 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 of 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. At the core of such an approach has to be a type checker that produces a complete set of type changes that would make the program type correct.

In this paper we present a method for *counter-factual 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 also implemented and evaluated a prototype for a type checker that is based on this technique.

To keep the complexity manageable we only produce so-called *atomic* type changes, that is, type changes for the leaves of the program's abstract syntax tree. This helps avoid the introduction of too exotic or too extreme changes. Consider, for example, the non-atomic type change suggested by Seminal for the `palin` program, which seems to be not realistic. Or consider changing a whole program to a value of type `Bool` or `Int`, which always works but is hardly ever correct.

However, errors that are best fixed by non-atomic expression changes are quite common. Examples are the swapping of function arguments or the addition of missing function arguments. The identification of such non-atomic program changes is not ruled out by

[5] ww2.cs.mu.oz.au/~sulzmann/chameleon/. Since Chameleon doesn't offer a type diagnosis option anymore, the result is reproduced directly from [24].

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

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

[8] 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/expression	To new type/expression	Result type
1	(1,19)	`(:)`[9]	`a -> [a] -> [a]`	`a -> [b] -> a`	`[a] -> Bool`
			`[z]`	`z`	
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`	`(a -> b -> a) -> a -> [b] -> [a]`	`([a] -> a -> [a]) -> [b] -> [a] -> [a]`	`[a] -> Bool`

Figure 1: Ranked list of single-location type and expression change suggestions inferred for the `palin` example.

the approach taken and can actually often be achieved by deducing expression changes from type changes.

Returning to the `palin` example, Figure 1 shows a ranked list of all (single-location) type changes, computed by our prototype, that can fix 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 atomic type changes, 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 deduced expression change.) We also show the position of the code in the program[10] 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 can't be remedied by exploiting type information of the program.

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 non-atomic 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, that is `z`, to replace the application `(:) z []`.

Note however, that we don't 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-change suggestions with atomic 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.

At the core of the proposed method 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 atomic type change suggestions.

The type system is based on a systematic variation of the types of atomic expressions 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 variation-based type-change inference can then be explained on a high level in Section 3.

The algorithmic aspects of type inference and some measures for controlling runtime complexity are discussed in Section 5. We also briefly describe a 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 counter-factual 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.

2. Variation-Based Typing

The idea of variation-based type change inference is to explicitly represent and reason about discrepancies between inferred and expected types that are detected by the type checker. This idea can be realized in different ways, and the counter-factual (CF) type

[9] Our prototype represents `[z]` as `(:) z []`.

[10] We have added the line and column numbers by hand since our prototype currently works on abstract syntax and doesn't have access to the information from the parser.

inference presented in this paper is just one incarnation of this more general strategy. In this section we will introduce the idea of variation-based typing, and gather some technical machinery that will then be employed to formalize counter-factual type change inference.

First, the goal of CF type inference 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 [11]. 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) [5]. 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 [11] 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 = `not` $A\langle\texttt{1},\texttt{True}\rangle$ contains the named choice A 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*. Selection takes a selector of the form $D.i$, where D is the name of the variation point, traverses the expression and replaces all the variation points named D with its corresponding ith alternative. 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 $A\langle\texttt{odd},\texttt{not}\rangle\ A\langle\texttt{1},\texttt{True}\rangle$ represents only the two expressions `odd 1` and `not True`, while $A\langle\texttt{odd},\texttt{not}\rangle\ B\langle\texttt{1},\texttt{True}\rangle$ 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 typing [5] solves this problem by introducing variational types and a method for typing variational programs in one run. The result of type checking a variational program is a variational type, from which the types for individual program variants can be obtained with the same selection as the program is derived.

The syntax of variational types is shown below where α ranges over type variables, and γ ranges over type constants.

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

The type $\bot$ is used to denote the occurrences of type errors and will be discussed shortly in Section 2.2.

Under variational typing, the expression $A\langle\texttt{odd},\texttt{not}\rangle$ has the type $A\langle\texttt{Int} \to \texttt{Bool},\texttt{Bool} \to \texttt{Bool}\rangle$. The most important property of variational typing is that the type for each plain expression selected from the variational expression can be obtained through the same selections from the corresponding variational type. For example, selecting $A.2$ from both the variational expression and type, the expression `not` has the type `Bool` $\to$ `Bool`.

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

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

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

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

$$\begin{array}{ll}
\phi \bowtie \phi \ = \top & D\langle\overline{\phi_1}\rangle \bowtie D\langle\overline{\phi_2}\rangle = D\langle\overline{\phi_1 \bowtie \phi_2}\rangle \\
\bot \bowtie \phi \ = \bot & D\langle\overline{\phi}\rangle \bowtie \phi' = D\langle\overline{\phi \bowtie \phi'}\rangle \\
\phi \bowtie \bot \ = \bot & \phi' \bowtie D\langle\overline{\phi}\rangle = D\langle\overline{\phi' \bowtie \phi}\rangle \\
\phi \bowtie \phi' \ = \bot & \phi_1 \to \phi_2 \bowtie \phi_1' \to \phi_2' = (\phi_1 \bowtie \phi_1') \otimes (\phi_2 \bowtie \phi_2')
\end{array}$$

$$\begin{array}{c|c}
\boxed{\lhd : \pi \times \phi \to \phi} & \boxed{\otimes : \pi \times \pi \to \pi} \\
\bot \lhd \phi = \bot & \top \otimes \pi = \pi \\
\top \lhd \phi = \phi & \bot \otimes \pi = \bot \\
D\langle\overline{\pi}\rangle \lhd \phi = D\langle\overline{\pi \lhd \phi}\rangle & D\langle\overline{\phi}\rangle \otimes \phi' = D\langle\overline{\phi \otimes \phi'}\rangle
\end{array}$$

Figure 2: Operations for typing applications

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 [11]. For example, $A\langle\texttt{Int},\texttt{Int}\rangle$ is equivalent to `Int`, written as $A\langle\texttt{Int},\texttt{Int}\rangle \equiv$ `Int`, since either decision in A yields `Int`. The equivalence relationship plays an important part in, and poses a challenge to, the unification of variational types. For details we refer to [5].

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 $A\langle\texttt{odd},\texttt{not}\rangle$ `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 [4] is to assign the type $\bot$ to program variants that contain type errors. The explicit representation of type errors via $\bot$ as normal 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, $A\langle\texttt{odd},\texttt{not}\rangle$ `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 [4].

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

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 2. Lifting specifically accounts for the case in 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 $A\langle \texttt{Int}, \perp\rangle \to A\langle \texttt{Bool}, \perp\rangle$.

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 2, a typing pattern is a (possibly deeply nested) choice of the two values $\perp$ (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 2 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 [4]. Note also that we employ the abbreviating notation $\bar{x}$ for a sequence of alternatives $x_1, \ldots, x_n$ (types or expressions) used within choices. (In Figure 2, this applies only to the case $n = 2$.)

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

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 2. We can essentially view it as the logical "and" operation if we treat $\top$ as true and $\perp$ as false. For example, when computing $\texttt{Int} \to A\langle \texttt{Bool}, \texttt{Int}\rangle \bowtie B\langle \texttt{Int}, \perp\rangle \to \texttt{Bool}$, we first obtain $B\langle \top, \perp\rangle$ and $A\langle \perp, \top\rangle$ for matching the argument types and return types, respectively. Next, we use $\otimes$ to derive the final result as $A\langle \perp, B\langle \top, \perp\rangle\rangle$.

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 $\perp$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 $\pi = A\langle B\langle \perp, \top\rangle, B\langle \top, \perp\rangle\rangle$, and masking the return type of the function, $\texttt{Bool}$, with π yields $A\langle B\langle \perp, \texttt{Bool}\rangle, B\langle \texttt{Bool}, \perp\rangle\rangle$.

3. Counter-Factual Typing

The main idea behind counter-factual 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 there are generally infinitely many changes that one could consider. Therefore, we perform the variation on the type level. Basically, we ask for each atomic expression e the counter-factual question: *What type should e have to make the program well typed?*

The counter-factual 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 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 doesn't. 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 $\texttt{not}$ and $\texttt{1}$, 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 $\texttt{not}$ and $\texttt{1}$ 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 [4] 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 [5].

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

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 $A\langle B\langle \perp, \top\rangle, \top\rangle$. Based on the unifier and the typing pattern, we can compute that the result type of the varied expression of $\texttt{not 1}$ is $\phi = A\langle B\langle \perp, \texttt{Bool}\rangle, \alpha_5\rangle$. From the result type and the unifier, we can draw the following conclusions.

- If we don't change e, that is, we select $A.1$ and $B.1$ from the varied expression, the type of the expression is $\perp$ (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 vary $\texttt{not}$ to some other expression f, that is, if we select variant $A.2$ and $B.1$ from the variational result 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 $\texttt{not}$ to an expression of type $\texttt{Int} \to \alpha_5$, $\texttt{not 1}$ becomes well typed. In the larger context, α_5 may be further constrained to have some other type.
- If we vary $\texttt{1}$ to some expression g, that is, if we select $A.1$ and $B.2$ from the variational type, then the result type becomes $\texttt{Bool}$.
- If we vary both $\texttt{not}$ to f and $\texttt{1}$ to g, which means to select $A.2$ and $B.2$, 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.

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	$::=$	$\gamma \mid \alpha \mid \tau \rightarrow \tau$
Variational types	ϕ	$::=$	$\tau \mid \bot \mid D\langle\phi,\phi\rangle \mid \phi \rightarrow \phi$
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	$::=$	$\varnothing \mid \Delta, (l, D\langle\phi,\phi\rangle)$

Figure 3: Syntax of expressions, types, and environments

This gives us all atomic 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.

4. Type-Change Inference

This section presents the type system that generates a complete set of atomic 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 3 shows the syntax for the expressions, types, and meta environments for the type system. We extend the bar notation to other sequences of elements, such as bindings. 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 nonunifiable 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.

$$\eta(\bot) = \bot \qquad \eta(\phi_1 \rightarrow \phi_2) = \eta(\phi_1) \rightarrow \eta(\phi_2)$$

$$\eta(\forall\overline{\alpha}.\phi) = \forall\overline{\alpha}.\eta_{/\overline{\alpha}}(\phi) \qquad \eta(\alpha) = \begin{cases} \alpha & \text{if } \alpha \notin dom(\eta) \\ \phi & \text{if } \alpha \mapsto \phi \in \eta \end{cases}$$

$$\eta(D\langle\overline{\phi}\rangle) = D\langle\overline{\eta(\phi)}\rangle$$

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

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

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

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

$$\textsc{Unbound}\ \frac{x \notin dom(\Gamma) \qquad D \text{ fresh}}{\Gamma \vdash x : D\langle\bot,\phi\rangle|\{(\ell(x), D\langle\bot,\phi\rangle)\}} \qquad \textsc{Abs}\ \frac{\Gamma, x \mapsto \phi \vdash e : \phi'|\Delta}{\Gamma \vdash \lambda x.e : \phi \rightarrow \phi'|\Delta}$$

$$\textsc{Let}\ \frac{\begin{array}{c}\Gamma, x \mapsto \phi \vdash e : \phi|\Delta \\ \overline{\alpha} = FV(\phi) - FV(\Gamma) \qquad \Gamma, x \mapsto \forall\overline{\alpha}.\phi \vdash e' : \phi'|\Delta'\end{array}}{\Gamma \vdash \texttt{let}\ x = e\ \texttt{in}\ e' : \phi'|\Delta \cup \Delta'}$$

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

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

Figure 4: Rules for type-change inference

$\forall\alpha_1\beta_1.D\langle\phi_1', \phi_2'\rangle$ 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 4 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 atomic subexpressions of e that will lead to the types in ϕ.

Since we are only interested in atomic changes during this phase, we only vary the leaves in the AST of programs, which are constants and variable references. 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 rules Abs and Let for abstractions and let-expressions are very similar to those 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.

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.

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

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

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

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

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

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

Figure 5: Simplifications and selection

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 atomic type changes. Based on this result we also show that the type-change inference system is a conservative extension of the Hindley-Milner type system (HM).

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 4 that is applicable 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 5 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 don't contribute to type errors. Thus, there is no need to simplify types nested in choice D in Figure 5. Also, we formally define the selection operation $\lfloor\phi\rfloor_s$ in Figure 5. 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 6. These typing rules are identical to an ordinary Hindley-Milner type system, except that they allow to "override" the types of atomic expressions 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 we use the "orelse" notation $\delta(e)||\tau$ to pick the type $\delta(e)$ if $\delta(e)$ is defined and τ otherwise.

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

$$\textsc{App-C}\ \frac{\Gamma;\delta \vdash e_1 : \tau_1 \to \tau \qquad \Gamma;\delta \vdash e_2 : \tau_1}{\Gamma;\delta \vdash e_1\ e_2 : \tau}$$

Figure 6: Rules for the type-update system

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 atomic change suggestions only.

We can now show that by applying any of the inferred type changes (using the rules in Figure 6), 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, $\{(\ell_1, A\langle\texttt{Int},\texttt{Bool}\rangle), (\ell_2, B\langle\texttt{Bool},\texttt{Int}\rangle)\}.2 = [A.2, B.2]$. Formally, we have the following result. (We assume that Δ has been simplified by $\Downarrow$ in Figure 5 and the alternatives of choices in Δ are plain, as mentioned before.)

THEOREM 1 (Type-change inference is consistent). *For any given e and Γ, if $\Gamma \vdash e : \phi|\Delta$ and there is some τ such that $\lfloor\phi\rfloor_{\Delta.2} = \tau$, then $\Gamma;\downarrow\Delta \vdash e : \tau$.*

Moreover, the 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 $\Gamma;\delta \vdash e : \tau$, then there exist ϕ, Δ, and a typing derivation for $\Gamma \vdash e : \phi|\Delta$ such that $\downarrow\Delta = \delta$ and $\lfloor\phi\rfloor_{\Delta.2} = \tau$.*

The proofs for these two theorems can be constructed through an induction over the typing derivations of both type systems. In particular, note that constants and variable reference have the same type in these two systems, regardless of whether or not they are changed.

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 don't make much sense. For example, we can derive $\Gamma \vdash \texttt{5} : A\langle\texttt{Int},\texttt{Bool}\rangle|\Delta$ where $\Delta = \{(\ell_5(\texttt{5}), A\langle\texttt{Int},\texttt{Bool}\rangle)\}$. However, since the expression `5` is type correct, it doesn't 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 `succ` to an expression of type $\texttt{Int} \to \texttt{Bool}$, or we can suggest to change `not` into something of type $\texttt{Int} \to \alpha_1$. 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

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

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 $\perp$ and `Bool`, respectively. Making the same selection into the third result type, we obtain `Bool`. Likewise, when we select with $[A.2, B.1]$, we get the types α_1, $\perp$, 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 $\phi|\Delta$ can be *more defined* than another result $\phi'|\Delta'$, which means that for any $\bar{s}$ for which $\lfloor \phi' \rfloor_{\bar{s}}$ yields a monotype then so does $\lfloor \phi \rfloor_{\bar{s}}$. Second, a result $\phi|\Delta$ can be *more general* than another result $\phi'|\Delta'$, written as $\phi \leq \phi'$, 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 $\delta_1 \leq \delta_2$, 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.

LEMMA 2 (Most defined type changes). *Given e and Γ and two typings $\Gamma \vdash e : \phi_1|\Delta_1$ and $\Gamma \vdash e : \phi_2|\Delta_2$, if $\lfloor \phi_1 \rfloor_{\bar{s}} = \perp$ 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* $\downarrow_{\bar{s}'}\Delta_3 = \downarrow_{\bar{s}'}\Delta_2$ *for all other* $\bar{s}'$.

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

LEMMA 3 (Generalizability of type changes). *For any two typings $\Gamma \vdash e : \phi_1|\Delta_1$ and $\Gamma \vdash e : \phi_2|\Delta_2$, if neither $\lfloor \phi_1 \rfloor_{\bar{s}} \leq \lfloor \phi_2 \rfloor_{\bar{s}}$ nor $\lfloor \phi_2 \rfloor_{\bar{s}} \leq \lfloor \phi_1 \rfloor_{\bar{s}}$ holds, there is a typing $\Gamma \vdash e : \phi_3|\Delta_3$ such that*

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

We can now combine and generalize Lemmas 2 and 3 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* $\lfloor \phi_1 \rfloor_{\bar{s}} = \perp$ *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* $\lfloor \phi_2 \rfloor_{\bar{s}} = \perp$ *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* $\lfloor \phi_3 \rfloor_{\bar{s}} \leq \tau_1$ *and* $\lfloor \phi_3 \rfloor_{\bar{s}} \leq \tau_2$. *Moreover,* $\downarrow_{\bar{s}}\Delta_3 \leq \downarrow_{\bar{s}}\Delta_1$ *and* $\downarrow_{\bar{s}}\Delta_3 \leq \downarrow_{\bar{s}}\Delta_2$.

The proofs for Lemma 2, Lemma 3 and Theorem 3 can be established by showing that in each derivation step, the result types from two derivations can always be combined into the result type for a third derivation such that the new result type is both more general and contains fewer errors. The key ingredient we need is that given a variational type, we can change the type for a specific variant and leave all other variants unchanged.

From Theorems 3 and 2 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 Γ, there is a typing $\Gamma \vdash e : \phi|\Delta$ such that for any δ if $\Gamma; \delta \vdash e : \tau$, then there is some $\bar{s}$ such that $\lfloor \phi \rfloor_{\bar{s}} \leq \tau$ and $\downarrow_{\bar{s}}\Delta \leq \delta$.*

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 $\Gamma \vdash e : \tau$ to express that expression e has the type τ under Γ in the HM type system.

THEOREM 5. *For any given e and Γ, $\Gamma; \varnothing \vdash e : \tau \Longleftrightarrow \Gamma \vdash e : \tau$.*

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 $\Gamma \vdash e : \tau$.*

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 [4], the inference algorithm is obtained by a straightforward translation of the typing rules presented in Figure 4. The cases for variable reference and `if` statements are shown below. Function application is very similar to `if` statements, and the cases for abstractions and let-expressions can be derived from $\mathcal{W}$ by simply adding the threading of Δ.

```
$infer : \Gamma \times e \rightarrow \theta \times \phi \times \Delta$
$infer(\Gamma, x) =$
    $\phi' \leftarrow inst(\Gamma(x))$          – returns $\perp$ when $x$ is unbound
    $\phi \leftarrow D\langle \phi', \alpha \rangle$                 – $D$ and $\alpha$ are fresh
    return $(\varnothing, \phi, \{(\ell(x), \phi)\})$

$infer(\Gamma,$ if $e_1$ then $e_2$ else $e_3) =$
    $(\theta_1, \phi_1, \Delta_1) \leftarrow infer(\Gamma, e_1)$
    $(\theta', \pi') \leftarrow vunify(\phi_1,$ Bool$)$
    $(\theta_2, \phi_2, \Delta_2) \leftarrow infer(\theta'\theta_1(\Gamma), e_2)$
    $(\theta_3, \phi_3, \Delta_3) \leftarrow infer(\theta_2\theta'\theta_1(\Gamma), e_3)$
    $(\theta_4, \pi_4) \leftarrow vunify(\theta_3(\phi_2), \phi_3,)$
    $\theta \leftarrow \theta_4\theta_3\theta_2\theta'\theta_1$
    return $(\theta, \pi' \lhd (\pi_4 \lhd \theta_4(\phi_3)), \theta(\Delta_1 \cup \Delta_2 \cup \Delta_3))$
```

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 *vunify*(ϕ_1, ϕ_2) for partial unification [4]. In addition to a partial unifier a typing pattern is generated to describe which variants are unified successfully and which aren't (see Section 3). Otherwise, the algorithm follows in a straightforward way the usual strategy for type inference.

We can prove that the algorithm *infer* correctly implements the typing rules in Figure 4, as expressed in the following theorems.

THEOREM 7 (Type-change inference is sound). *Given any e and Γ, if 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 $\lfloor\phi_2\rfloor_{\bar{s}} = \perp$ or $\lfloor\phi_1\rfloor_{\bar{s}} \leq \lfloor\phi_2\rfloor_{\bar{s}}$. 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 $\phi_1 \leq \phi_2$ holds.

THEOREM 8 (Type-change inference is complete and principal). *If $\theta(\Gamma) \vdash e : \phi|\Delta$, then 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 occurs only with deep nestings of 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 $h : \gamma \to \gamma \to \ldots \to \gamma$ because only the function f requires all argument types to be unified, which causes choice nesting to happen.

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: (A) choices whose alternatives are unifiable, and (B) choices whose alternatives contain errors in the same places. These two conditions ensure that the eliminated choices are unlikely to contribute to type errors. There are cases in which this strategy fails 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 aren't. 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 [5].

Finally, we briefly describe a set of simple heuristics that define the ranking of type and expression changes. (1) We prefer places that have deduced expression changes (see Section 6) because these changes reflect common editing mistakes [18]. (2) We favor changes that are lower in the abstract syntax trees because changes at those places have least effect on the context and are least likely to introduce exotic results. (3) We prefer changes that have minimal shape difference between the inferred type and the expected type. For example, a change that doesn't influence the arities of function types is ranked higher than a change that does change arities.

6. Deducing Expression Changes

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.[11]

```
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 [21] 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 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. 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 are 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 more such ad-hoc changes that are useful in some situations, but we will not discuss them here.

In Section 8 we will compare our method with McAdam's original. 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 counter-factual typing approach, we have implemented a prototype of type-change

[11] This example is adapted from [18], where `zipWith` is called `map2`.

	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

Figure 7: Evaluation results for different approaches over 121 collected examples (in %).

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 [18, 19], Helium [13, 14], 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 don't suggest changes.[12]

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 [14, 21, 27, 29] and papers that represent most recent and older work [15, 18, 23]. 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) doesn't currently support type classes. This exclusion did not have a significant effect. We gathered 8 unique examples regarding type classes involved in type errors discussed in [24, 24, 27]. 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 [27].

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 [2], Helium suggests to change `True` to something of 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 7 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

[12] There are a few interactive approaches that have been proposed [6, 24], but they do currently not provide running implementations. Moreover, Chameleon [27] 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.

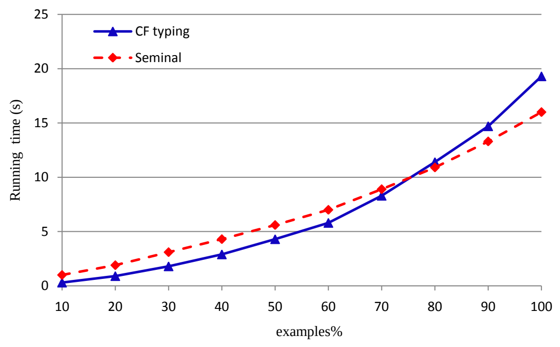

Figure 8: Running time for typing x% of the examples 10 times.

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 which 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. In several cases, Seminal's change suggestion doesn't fix the type error.

Most cases for which CF typing fails are caused by missing parentheses. For example, for the expression `print "a" ++ "b"` [18], 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.

With the help of variational typing, we can generate all the potential changes very efficiently. The running time for all the collected examples is within 2 seconds. Figure 8 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 same machine with Cygwin 5.1. The purpose of the graph is simply to demonstrate the feasibility of our approach.

Second, we have evaluated how increasing levels of choice nestings affect the efficiency of the inference algorithm and how putting a limit on maximum nesting levels as described in Section 5 can regain efficiency at the cost of precision. For this purpose, we have automatically generated large examples, and we use functions of

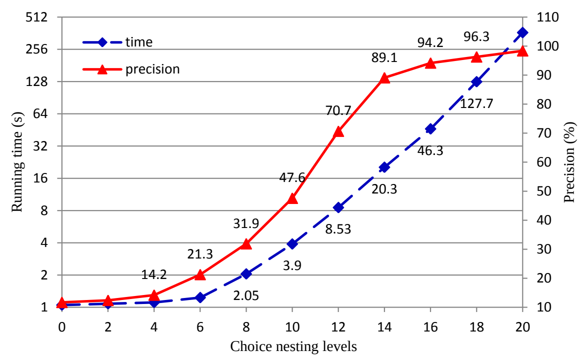

Figure 9: Limits on choice nesting trade efficiency for precision.

types like $\alpha \rightarrow \alpha \rightarrow \ldots \rightarrow \alpha$ 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 60000 nodes in its tree representation.

Figure 9 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 dividing the number of examples that have correct change suggestions over 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 $\mathcal{W}$ and report an error as soon as the algorithm fails. Since the original algorithm $\mathcal{W}$ 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 $\mathcal{M}$ [17], $\mathcal{G}$ [10], $\mathcal{W}^{SYM}$ and $\mathcal{M}^{SYM}$ [20], and $\mathcal{UAE}$ and $\mathcal{IEI}$ [30]. All these algorithms interpret the place of unification failure as the source of the type error. In contrast, Johnson and Walz [15] and the Helium tool [13, 14] 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.

In deducing expression changes from type changes, we have used (an extension of) McAdam's technique [21]. Since his approach is based on the algorithm $\mathcal{W}$, it suffers from the bias of error locating mentioned above. Moreover, his approach doesn't 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 [26] records each unification step so that they can be tracked back to the failure point. Duggan and Bent [9] on the other hand record the reason for each unification that is being performed. Beaven and Stansifer [1] and Yang [28] 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 algorithms work 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's not always clear about how to exploit it and change the program.

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 enable a type debugging paradigm that is based on the idea of replacing a suspicious program snippet by a fresh variable [2]. If such 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 do these steps manually. Later, Braßel [3] automated this process by systematically commenting out parts of the program and running the type checker iteratively. Since type changing is based on unification, it can again introduce the bias problem. Also, it is unclear how to handle programs that contain more than one type error.

Through employing a number of different techniques, Chitil [6], Neubauer and Thiemann [22], and Stuckey [24, 27] have developed tools that allow users to explore a program and inspect the types for any subexpression. Chameleon [24, 27] 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.

Error slicing The main advantage of slicing approaches [12, 23, 25] is that they return all locations related to type errors. The downside is that they cover too many locations. Recent improvements in Chameleon [27] 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 of how to get rid of the type error.

Like error slicing approaches, our CF typing approach is complete in not missing any potential change. However, the changes we presented to users involve fewer locations. Usually, users have to focus on only one location and its suggestions for type changes.

Embracing type uncertainty Similar to choice types, sum types can also encode many types. Neubauer and Thiemann [22] 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. Technically, named choice types 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 doesn't provide specific change locations or change suggestions.

Typing by searching CF typing and Seminal [18, 19] 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, which makes it impossible for Seminal to generate a complete set of type-change suggestions. 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. This way of searching causes Seminal to make mistakes in locating errors when the first part of the program itself doesn't contain a type error but actually triggers type errors because it's 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.

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. Compared to our change deduction approach, this has both advantages and disadvantages. In some cases, it can find a correct change while our approach fails to do so, as, for example, in the missing-parentheses problem discussed in Section 7. On the other hand, its power to generate arbitrarily complicated changes can lead to bizarre suggestions, such as the suggestion to change `xs == (rev xs)` to `(==) (xs,(rev xs))`.

9. Conclusions

We have presented a new method for debugging type errors. The approach is based on the notion of counter-factual typing, which is the idea of systematically varying the types of all atomic program elements 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, the integrated choice-based representation provides opportunities for defining type queries that can be used to examine programs and also form the basis for sophisticated user interfaces to support more interactive forms of type debugging. We also plan to investigate how well the approach works for the debugging of type errors in richer type systems.

References

[1] M. Beaven and R. Stansifer. Explaining type errors in polymorphic languages. *ACM Letters on Programming Languages and Systems*, 2:17–30, 1994.

[2] K. L. Bernstein and E. W. Stark. Debugging type errors. Technical report, State University of New York at Stony Brook, 1995.

[3] B. Braßel. Typehope: There is hope for your type errors. In *Int. Workshop on Implementation of Functional Languages*, 2004.

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

[5] S. Chen, M. Erwig, and E. Walkingshaw. Extending Type Inference to Variational Programs. *ACM Trans. on Programming Languages and Systems*, 2013. To appear.

[6] O. Chitil. Compositional explanation of types and algorithmic debugging of type errors. In *ACM Int. Conf. on Functional Programming*, pages 193–204, September 2001.

[7] V. Choppella. *Unification Source-Tracking with Application To Diagnosis of Type Inference*. PhD thesis, Indiana University, 2002.

[8] L. Damas and R. Milner. Principal type-schemes for functional programs. In *ACM Symp. on Principles of Programming Languages*, pages 207–212, 1982.

[9] D. Duggan and F. Bent. Explaining type inference. In *Science of Computer Programming*, pages 37–83, 1995.

[10] H. Eo, O. Lee, and K. Yi. Proofs of a set of hybrid let-polymorphic type inference algorithms. *New Generation Computing*, 22(1):1–36, 2004.

[11] M. Erwig and E. Walkingshaw. The Choice Calculus: A Representation for Software Variation. *ACM Trans. on Software Engineering and Methodology*, 21(1):6:1–6:27, 2011.

[12] C. Haack and J. B. Wells. Type error slicing in implicitly typed higher-order languages. In *European Symposium on Programming*, pages 284–301, 2003.

[13] B. Heeren, D. Leijen, and A. van IJzendoorn. Helium, for learning haskell. In *Proceedings of the 2003 ACM SIGPLAN workshop on Haskell*, Haskell '03, pages 62–71, New York, NY, USA, 2003. ACM.

[14] B. J. Heeren. *Top Quality Type Error Messages*. PhD thesis, Universiteit Utrecht, The Netherlands, Sept. 2005.

[15] G. F. Johnson and J. A. Walz. A maximum-flow approach to anomaly isolation in unification-based incremental type inference. In *ACM Symp. on Principles of Programming Languages*, pages 44–57, 1986.

[16] O. Lee and K. Yi. Proofs about a folklore let-polymorphic type inference algorithm. *ACM Trans. on Programming Languages and Systems*, 20(4):707–723, July 1998.

[17] O. Lee and K. Yi. A generalized let-polymorphic type inference algorithm. Technical report, Technical Memorandum ROPAS-2000-5, Research on Program Analysis System, Korea Advanced Institute of Science and Technology, 2000.

[18] B. Lerner, M. Flower, D. Grossman, and C. Chambers. Searching for type-error messages. In *ACM Int. Conf. on Programming Language Design and Implementation*, pages 425–434, 2007.

[19] B. Lerner, D. Grossman, and C. Chambers. Seminal: searching for ml type-error messages. In *Workshop on ML*, pages 63–73, 2006.

[20] B. J. McAdam. *Repairing type errors in functional programs*. PhD thesis, University of Edinburgh. College of Science and Engineering. School of Informatics., 2002.

[21] B. J. McAdam. *Reporting Type Errors in Functional Programs*. PhD thesis, Larboratory for Foundations of Computer Science, The University of Edinburgh, 2002.

[22] M. Neubauer and P. Thiemann. Discriminative sum types locate the source of type errors. In *ACM Int. Conf. on Functional Programming*, pages 15–26, 2003.

[23] T. Schilling. Constraint-free type error slicing. In *Trends in Functional Programming*, pages 1–16. Springer, 2012.

[24] P. J. Stuckey, M. Sulzmann, and J. Wazny. Interactive type debugging in haskell. In *ACM SIGPLAN Workshop on Haskell*, pages 72–83, 2003.

[25] F. Tip and T. B. Dinesh. A slicing-based approach for locating type errors. *ACM Trans. on Software Engineering and Methodology*, 10(1):5–55, Jan. 2001.

[26] M. Wand. Finding the source of type errors. In *ACM Symp. on Principles of Programming Languages*, pages 38–43, 1986.

[27] J. R. Wazny. *Type inference and type error diagnosis for Hindley/Milner with extensions*. PhD thesis, The University of Melbourne, January 2006.

[28] J. Yang. Explaining type errors by finding the source of a type conflict. In *Trends in Functional Programming*, pages 58–66. Intellect Books, 2000.

[29] J. Yang. *Improving Polymorphic Type Explanations*. PhD thesis, Heriot-Watt University, May 2001.

[30] J. Yang, G. Michaelson, P. Trinder, and J. B. Wells. Improved type error reporting. In *Int. Workshop on Implementation of Functional Languages*, pages 71–86, 2000.