Better Type-Error Messages Through Lazy Typing *

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

Producing precise and helpful error messages for type inference is still a challenge for implementations of functional languages. Current approaches often lack precision in terms of locating the origins of type errors. Moreover, suggestions for how to fix type errors that are offered by some tools are also often vague or incorrect.

To address this problem we have developed a new approach to identifying type errors that is based on delaying typing decisions and systematically gathering context information to support the delayed decision making. Our technique, which we call *lazy typing*, is based on explicitly representing conflicting types and type errors in *choice types* that will be accumulated during the typing process. The structure of these types is then analyzed to produce error messages and, in many cases, also type-change suggestions.

We will demonstrate that lazy typing is often more precise in locating type errors than existing tools and that it can also produce good type-change suggestions. We do not consider lazy typing as a replacement for other techniques, but rather as an addition that could help improve other approaches.

1. Introduction

One of the major challenges faced by current implementations of type inference algorithms is the production of error messages that help programmers fix mistakes in the code. Typical problems include the reporting of error messages in terms of internal compiler representations, or showing error messages at locations that are distant from the source of the error.

Quite a few solutions have been proposed to locate type errors. One approach is to eliminate the left-to-right bias of type inference [13, 14, 17] by viewing the subexpression symmetrically or traversing the expressions in a top-down manner. Another approach is to report several program sites that most likely contribute to the type inconsistency [23, 24, 27, 28] instead of committing to only one error location. A related approach is to use program slicing to determine all the positions that are related to the type errors [5, 6, 9, 22, 25].

The need for improving type-error diagnosis was recognized soon after the original Hindley-Milner type system was proposed [11, 18, 26]. However, despite the considerable research efforts devoted to this problem, and the improvements that were achieved, each of the proposed solutions has its own shortcomings and performs poorly for certain programs.

Consider, for example, the following function that splits a list into two lists by placing elements at odd positions in the first list and those at even positions into the second.

```
split xs = case xs of
             []       -> ([], [])
             [x]      -> ([], x)
             (x:y:zs) -> let (xs, ys) = split zs
                         in (x:xs, y:ys)
```

Even though the type error in this definition is not hard to spot, existing type inference systems have a hard time locating it precisely. In the following we will show the error messages from several tools. For presentation purpose, we have edited the outputs slightly by changing their indentation and line breaks.

The Glasgow Haskell Compiler (GHC) 7.6,[1] which is the most widely used Haskell compiler, prints the following message.

```
Occurs check: cannot construct the infinite type: a0 = [a0]
In the first argument of ‘(:)’, namely ‘y’
In the expression: y : ys
In the expression: (x : xs, y : ys)
```

The problem with GHC (and also with Hugs98[2]) is that the generated error message does not directly point to where the source of the error is or how it could be fixed. The error message talks in terms of the compiler, using the internal representation of types and giving reasons why unification fails in compiler jargon. Even for experienced programmers, such error messages require some effort to manually reconstruct some of the types and solve unification problems.

The idea of error slicing tools is to find *all* program positions that contribute to a type error. A typical problem with the slicing approach is that the produced error messages are too general and cover too many locations. A programmer still has to find the real cause of the type error, which again may require a lot of effort. This issue has been addressed by pursuing techniques to minimize the possible locations contributing to a type error. As an example, the Chameleon Type Debugger[3] shows the following output.

* This work is supported by the the National Science Foundation under the grants CCF-0917092 and CCF-1219165.

[Copyright notice will appear here once 'preprint' option is removed.]

[1] www.haskell.org/ghc/

[2] www.haskell.org/hugs/

[3] ww2.cs.mu.oz.au/~sulzmann/chameleon/

```
ERROR: Type error
Problem : Case alternatives incompatible
Types   : [a] -> (b,a)
          [a] -> (c,[a])
Conflict: split xs = case xs of
            [] -> ([],[])
            [x] -> ([],x)
            (x:y:zs) ->
              let (xs, ys) = split zs
              in (x:xs, y:ys)
```

Chameleon is a clear improvement over other error slicing tools. It achieves this by transforming programs into CHR and through the use of a constraint solver to extract minimal unsatisfiable constraints, from which involved program locations are derived. Still, a programmer must investigate several places to precisely locate the type error. This approach is very useful when the error locations are very close, but less helpful when they are far apart. Moreover, deciding which types should be at specific locations to solve the type conflict still requires some effort.

The Helium compiler[4] was especially developed to support the teaching of typed functional programming languages and has a focus on generating good error messages [10]. It produces the following message.

```
(5,36): Type error in constructor
 expression       : :
   type           : a   -> [a] -> [a]
   expected type  : [b] -> [b] -> [b]
 because          : unification would give infinite type
 probable fix     : use ++ instead

Compilation failed with 1 error
```

Although the proposed change from `(:)` to `(++)` fixes the type error, this change also causes the types of the other two alternatives to change. In general, it seems preferable to change the definition rather than the use, and, if possible, to minimize the effect of the change.

As another example, consider the following function to compute Fibonacci numbers. This program contains a type error since the return types of case alternatives for the function `f` are different.

```
f x = case x of
       0 -> [0]
       1 -> 1

fib x = case x of
      0 -> f x
      1 -> f x
      n -> fib (n-1) + fib (n-2)
```

GHC produces the following message.[5]

```
Couldn't match expected type `[t0]' with actual type `Int'
In the expression: fib (n - 1) `plus` fib (n - 2)
In a case alternative: n -> fib (n - 1) `plus` fib (n - 2)
In the expression:
  case x of {
    0 -> f x
    1 -> f x
    n -> fib (n - 1) `plus` fib (n - 2) }
```

The Helium compiler shows the following error message.

```
Compiling ./Fib.hs
(3,13): Type error in right-hand side
expression       : 1
  type           : Int
  does not match : [Int]

Compilation failed with 1 error
```

Seminal[6] is a tool that searches for type-error messages [15, 16] in ML programs. It attributes the type error to the expression `1` and suggests the following possible fixes. We observe that neither the suggested error location is accurate, nor will the proposed change fix the type error.

```
File "fib.ml", line 3, characters 11-12:
This expression has type int but is here used
with type int list
Relevant code: 1
----------------------------------------
File "fib.ml", line 3, characters 11-12:
Try replacing
    1 with 1; [[...]]
of type
    int list
within context
    let f x = (match x with
                | 0 -> [0]
                | 1 -> (1; [[...]])) ;;
```

The problem with change-suggesting tools is that sometimes the messages are misleading by pointing to the wrong error location(s), and the suggested program repairs don't fix the type error. For both of the above examples, the change-suggesting tools Helium and Seminal fail to point to the erroneous source code and will not fix the type error.

1.1 Delayed Type Constraint Solving

How can we improve the low precision of error localization? It turns out that better exploiting the context of expressions containing type errors are detected can support error localization and also lead to better change suggestions. Specifically, context information can be exploited by delaying the decision about a type of an expression and making it dependent on the type information gathered from the context. In cases when the expression turns out to exhibit a type error, the availability of the type information for the context can help to decide what is wrong with the expression and point more precisely to the source of the type error.

The basic idea of the delaying strategy is to turn an equality constraint between types, such as $\tau = \tau'$, into a *choice* between the two types, which we write as $A\langle\tau,\tau'\rangle$ (we'll explain this notation in more detail shortly). Instead of enforcing the constraint, which potentially causes an immediate failure of type checking, we continue the type inference process with the two possibilities τ and τ'. If $\tau \neq \tau'$, the inference will eventually fail too, but at a later point when additional context information is available. We call this strategy *lazy typing*.

In this paper we restrict the application of the lazy typing strategy to the typing of conditionals and case expressions. This is sufficient to illustrate the technique and its potential for improving type error localization.

In general, lazy typing works as follows. Consider an expression e that is required by some typing rule to have the type τ_1 as well as the type τ_2. If $\tau_1 \neq \tau_2$, we know that the expression is not type correct, but we generally do not know which of the two types is the "wrong one" that is the cause of the type error. Now let's assume that the context E in which e occurs expects the type τ_1,

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

[5] GHC 7.6 will give `fib` the type `(Eq a, Num a, Num [t], Num t) => a -> [t]`, which is not what we want. To avert this problem, we can define a function `plus` that has the type `Int -> Int -> Int` and use that instead of `+`.

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

that is, $E[e]$ is type correct when e is of type τ_1 and type incorrect if e is of type τ_2. This indicates that e's type really should be τ_1 and that the expression responsible for requiring e's type to be τ_2 is to be blamed for the type error and should be reported as the error location. This works because in general it is unlikely that E accidentally works with τ_2 while in fact it only works with τ_1. The confidence in such context-derived judgments increases when e occurs in several different E's that all agree on their type expectation.

In the `fib` example, the function `f` has a type conflict because the two alternatives return different types. By only looking at `f` we can't tell which type is preferable over the other. However, when examined in the context of the function `fib`, we recognize that `f` is type correct when the case alternatives have the type `Int`. Thus, we can conclude that the first branch that gives rise to the type `[Int]` causes the type error.

1.2 Lazy Typing Using Choice Types

To set up the technical development in the rest of this paper we illustrate how the technique of lazy typing with choice types works with the examples presented earlier.

In the `fib` example, the typing rule for the case expression requires the case expression in `f` to have the type `[Int]` as well as `Int`. Instead of reporting a type error right away, we delay the resolution of this discrepancy by temporarily keeping both types and assigning the case expression the following *choice type* [2].

$$A\langle \texttt{[Int]}, \texttt{Int}\rangle$$

A choice type consists of a *dimension name* for naming a variation point (here A) and all the types that constitute the alternatives of the choice (here `[Int]` and `Int`). Each dimension represents a decision that can be made about a type. A choice type of n alternatives represents n types, and a plain type can be obtained by eliminating choices through the selection of particular alternatives. Note that choice types can occur as subexpressions of type expression. For example, the type of `f` in the `fib` example is given by the following type expression, which describes the two type $\texttt{Int} \to \texttt{[Int]}$, obtained when the first alternative of A is selected, and $\texttt{Int} \to \texttt{Int}$, obtained as the second alternative.

$$\texttt{Int} \to A\langle \texttt{[Int]}, \texttt{Int}\rangle$$

In general, type expressions will contain multiple variation points. While for each newly delayed decision a choice type with a fresh dimension name will be created, the sharing of types through type assumptions in the type environment as well as the propagation of choice types through typing rules can lead to separate choice types that share the same dimension. The alternatives in those choice types are synchronized in the sense that the selection of the kth alternative in the dimension A will cause all kth alternatives of all choices labeled A to be selected.

Since `fib` is a recursive function, the typing process will involve the finding of fixpoints. For the case expression in `fib`, we create a choice type in a new dimension B with three alternatives that represent the types of the three branches. The first two alternatives are $A\langle \texttt{[Int]}, \texttt{Int}\rangle$ because applying `f` to an `Int` value yields a value of type $A\langle \texttt{[Int]}, \texttt{Int}\rangle$. Thus we obtain the following tentative typings, where β is a fresh type variable.

$$\begin{aligned} \texttt{fib} &: \texttt{Int} \to B\langle A\langle \texttt{[Int]}, \texttt{Int}\rangle, A\langle \texttt{[Int]}, \texttt{Int}\rangle, \beta\rangle \\ \texttt{fib (n-1)} &: B\langle A\langle \texttt{[Int]}, \texttt{Int}\rangle, A\langle \texttt{[Int]}, \texttt{Int}\rangle, \beta\rangle \end{aligned}$$

Now partially applying `+` to `fib (n-1)` leads to the need to unify `Int` and $B\langle A\langle \texttt{[Int]}, \texttt{Int}\rangle, A\langle \texttt{[Int]}, \texttt{Int}\rangle, \beta\rangle$. This unification problem can be solved by extending unification across choice types [2, 3]. The process can be illustrated as follows.

$$\begin{aligned} & \texttt{Int} \equiv^? B\langle A\langle \texttt{[Int]}, \texttt{Int}\rangle, A\langle \texttt{[Int]}, \texttt{Int}\rangle, \beta\rangle \\ = {} & B\langle \texttt{Int} \equiv^? A\langle \texttt{[Int]}, \texttt{Int}\rangle, \texttt{Int} \equiv^? A\langle \texttt{[Int]}, \texttt{Int}\rangle, \texttt{Int} \equiv^? \beta\rangle \\ = {} & B\langle A\langle \texttt{Int} \equiv^? \texttt{[Int]}, \texttt{Int} \equiv^? \texttt{Int}\rangle, A\langle \ldots\rangle, \texttt{Int} \equiv^? \beta\rangle \end{aligned}$$

At this point we can see that the two instances of the unification problem $\texttt{Int} \equiv^? \texttt{[Int]}$ will fail (the other problems all result in the type `Int`). Since we use choice types to delay typing decisions, we will represent this failure explicitly using the notion of an *error type*, written as $\bot$, instead of aborting the type inference process. In fact, the explicit representation of the unification failure will later help us with the localization of the source of the type error. Thus we obtain the following type as a result of the unification problem.

$$\tau_B = B\langle A\langle \bot, \texttt{Int}\rangle, A\langle \bot, \texttt{Int}\rangle, \texttt{Int}\rangle$$

Therefore, we obtain the following type for `(+) (fib (n-1))`.

$$\texttt{(+) (fib (n-1))} : B\langle A\langle \bot, \texttt{Int}\rangle, A\langle \bot, \texttt{Int}\rangle, \texttt{Int}\rangle \to \texttt{Int}$$

Conceptually, we can view that the unification process refines the type for `fib` to change to $\texttt{Int} \to B\langle A\langle \bot, \texttt{Int}\rangle, A\langle \bot, \texttt{Int}\rangle, \texttt{Int}\rangle$ (we can also achieve this by iterating the typing of `fib` for several times). The application of `(+) (fib (n-1))` to `fib (n-2)` requires the unification of τ_B, the type for `fib (n-1)`, with itself, which does not lead to any further instantiations or changes, that is, `fib` has the following type.

$$\texttt{fib} : \texttt{Int} \to B\langle A\langle \bot, \texttt{Int}\rangle, A\langle \bot, \texttt{Int}\rangle, \texttt{Int}\rangle$$

Note that the type $A\langle \bot, \texttt{Int}\rangle$ is a refinement of the choice type originally created for `f`, which already indicates that the source of the type error is in the first branch of the case expression for `f`.

A variational type derived by lazy typing contains information about the location of the type error and, in addition, also often allows the generation of change suggestions.

In the `fib` example the occurrence of the error type suggests that the first alternative of choice A causes the type error, which is the type for the first case alternative for `f`. We thus report `[0]` as the cause of the type error. Moreover, we can also suggest that the expected type should be `Int`, the same as the type for the second case alternative, because it is compatible with the rest of the program.

Our prototype implementation reports the type error and the change suggestion in the following way.[7]

```
(2,13): Type error in expression:
  [0]
Of type:  [Int]
Should have type:  Int
```

In cases when the type of the erroneous expression is ambiguous, we report the different alternative types to illustrate the conflicts in the program. For example, for a top-level conditional with incompatible branches (see also program `if1` in Appendix B), we produce the following error message. Also, in this example, since we cannot confidently suggest a type change, we do not produce a suggestion.

```
(1,4): Type error in expression:
  if True then (\f-> f (f 2)) else (\g-> g (g True))
Of type:  (Int->Int)->Int
     or:  (Bool->Bool)->Bool
```

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

For the `split` example lazy typing proceeds in a very similar way and produces a ternary choice type for the case expression. The initial type assumption for `split` is the following.

$$\texttt{split} : [\alpha] \to A\langle([\alpha_1],[\alpha_2]),([\beta_1],\alpha),([\alpha],[\alpha])\rangle$$

Matching the recursive call `split zs` against the pair `(xs,ys)` causes, through unification and the introduction of a type error, the refinement of the type to the following.

$$\texttt{split} : [\alpha] \to A\langle([\alpha],[\alpha]),([\alpha],\bot),([\alpha],[\alpha])\rangle$$

This is also the result type. The location of the error type in the choice type puts the blame for the type error on the second alternative of the case expression. The suggested change is derived from the other alternatives in the choice type.

```
(3,31): Type error in expression:
  x
Of type:  a
Should have type:  [a]
```

The idea of lazy typing seems to be similar to the concept of discriminative sum types [19, 20], in which two types are combined into a sum type when an attempt to unify them fails. However, there are several important differences. First, the choice types are named and thus provide more find-grained control over the grouping of types, unification, and unification failures. Sum types are *always* unified componentwise, whereas we do this only for choice types under the same dimension name. For choice types with different dimension names, each alternative of a choice type is unified with all alternatives of the other choice type. Second, the two type systems impose different orders for unifications. For sum types, different branches are unified before unification with the context, whereas for choice types, branches are unified with the context first, followed by the unification of all result types. Finally, in choice types we make the occurrences of type errors explicit through $\bot$-types, which makes it easier to locate errors. As will be discussed in Section 6, all these differences lead to significant differences in the behavior and results of the two type systems.

1.3 Contributions

In Section 2 we review the notion of choice types [2, 3] and their unification plus the required background of the formal underpinning, the choice calculus [8]. This will set the stage for the use of choice types as an underlying representation for lazy typing. The organization of the rest of the paper is described based on the contributions made in the paper.

- We define a type system for lambda calculus with let polymorphism that realizes lazy typing in Section 3. The use of choice and error types allows us to delay the resolution of type equality constraints. We show that the type system is correct in the sense of being a conservative extension of the standard Hindley-Milner system (Theorem 1, Corollary 1, and Corollary 2).
- In Section 4 we present an algorithm for the localization of type errors and the derivation of change suggestions. The algorithm works by analyzing the structure of choice types and delayed decisions that have been produced by the type system. Error localization finds the most likely error locations by identifying minimal incompatibility with context. Suggestion of type changes shows a very high success rate since each suggestion is guaranteed to remove at least one type error. (In cases where not enough information is available to achieve this, we refrain from making any suggestion.)
- We evaluate the potential impact and benefits of lazy typing in Section 5 by comparing the results produced by our type system and error localization algorithm with existing systems. We find that lazy typing correctly reports error locations in most cases, and all of its type suggestions are correct. There are some situations in which lazy typing is unable to find a suggestion. The comparison with other tools shows that lazy typing adds new value to the reporting of type errors.

2. Variational Partial Types for Program Families

This section provides the technical background required for the definition of the type system for lazy typing. Specifically, we introduce the notions of variational types, partial types, and their equivalence.

In our previous work, we have introduced variational types to type variational programs [3]. Variation in expressions and types is represented by named choices that represent decision points in expressions and types. Decisions about variations are made by applying a selection operation that picks the kth option in a particular dimension D, which causes the replacement of each dimension named D by its kth alternative.

For example, the variational expression $e = \lambda\texttt{x}.A\langle\texttt{x+1},\texttt{[x-1]}\rangle$ represents two expressions that can be obtained by selecting $A.1$ and $A.2$, respectively. The type of e varies for the two alternatives and can also be expressed using a choice as $\texttt{Int} \to A\langle\texttt{Int},\texttt{[Int]}\rangle$.

Next consider the following slight extension of e to $e' = \lambda\texttt{x}.A\langle\texttt{x+1},\texttt{[x-1]}\rangle\texttt{*2}$. Here only the first alternative of e' is type correct, which is reflected in the partial type $e' : \texttt{Int} \to A\langle\texttt{Int},\bot\rangle$.

In this paper we are in a sense *inverting* the direction of this view: First, given a plain expression e, we try to infer a plain type for it. If this fails, we introduce choice types to account for discrepancies in types that are expected to be equal. These choice types essentially "hypothesize" choices in e that would lead to the generated choice types. Consider again the type $\tau_{\texttt{f}} = \texttt{Int} \to A\langle\texttt{[Int]},\texttt{Int}\rangle$ that was inferred for `f`. We can imagine a changed definition for `f` in which the expression `[0]`, the identified source of the type error, is replaced by a choice expression $A\langle\texttt{[0]},e\rangle$ where e is of type `Int`. For this definition, the variational type system would infer exactly the type $\tau_{\texttt{f}}$. The type system presented in this paper infers $\tau_{\texttt{f}}$ and uses the type and error information to interpret the hypothesized choice $A\langle\texttt{[0]},e\rangle$ as a suggestion to replace `[0]` by an expression e of type `Int`.

The syntax of variational types is as follows.

$$\tau \;::=\; \alpha \mid \tau \to \tau \mid D\langle\tau,\tau\rangle \mid \bot$$

Here α ranges over type variables, and $\tau \to \tau$ represents the usual function types. The choice type $D\langle\tau,\tau\rangle$ has an associated dimension name D, and $\bot$ represents type errors.

We have already explained that choice types can be arbitrarily nested and choice types in the same dimension are synchronized through the selection operation, which takes the kth alternative from each choice in the same dimension.

Choice types pose a challenge to the unification algorithm since choices are subject to equivalence rules that requires the unification to work modulo an equational theory (shown in Figure 1). We illustrate the equivalences with a few examples here, but defer a full exposition of this topic to [3]. A simple example is the choice type $A\langle\texttt{Int},\texttt{Int}\rangle$, which is the same as `Int` since either decision in A yields `Int`. Thus $A\langle\texttt{Int},\texttt{Int}\rangle$ and `Int` are equivalent, a fact that is written as $A\langle\texttt{Int},\texttt{Int}\rangle \equiv \texttt{Int}$. This relationship is expressed by the rule C-IDEMP, which says that a choice type can be replaced by one of its alternatives if they all are equivalent.

Type equivalence is symmetric, reflexive and transitive, and it is inductive over choices constructor and the function type constructor. The rule F-C shows that the arrow constructor can be pushed down over choice types, and vice versa. The rules C-C-SWAP1 and C-C-SWAP2 show that choice nestings can be reordered. The rules C-C-MERGE1 and C-C-MERGE2 express the idea of choice dominance, which essentially means to remove dead alternatives in

$$\textsc{Refl}\ \ \tau \equiv \tau \qquad \textsc{Symm}\ \ \frac{\tau \equiv \tau'}{\tau' \equiv \tau} \qquad \textsc{Trans}\ \ \frac{\tau \equiv \tau' \qquad \tau' \equiv \tau''}{\tau \equiv \tau''}$$

$$\textsc{Choice}\ \ \frac{\tau_1 \equiv \tau_1' \qquad \tau_2 \equiv \tau_2'}{D\langle \tau_1, \tau_2 \rangle \equiv D\langle \tau_1', \tau_2' \rangle} \qquad \textsc{Fun}\ \ \frac{\tau_l' \equiv \tau_r' \qquad \tau_l \equiv \tau_r}{\tau_l' \to \tau_l \equiv \tau_r' \to \tau_r}$$

$$\text{F-C}\ \ D\langle \tau_1, \tau_2 \rangle \to D\langle \tau_1', \tau_2' \rangle \equiv D\langle \tau_1 \to \tau_1', \tau_2 \to \tau_2' \rangle$$

$$\text{C-Idemp}\ \ \frac{\tau_1 \equiv \tau \qquad \tau_2 \equiv \tau}{D\langle \tau_1, \tau_2 \rangle \equiv \tau}$$

$$\text{C-C-Swap1}\ \ D'\langle D\langle \tau_1, \tau_2 \rangle, \tau_3 \rangle \equiv D\langle D'\langle \tau_1, \tau_3 \rangle, D'\langle \tau_2, \tau_3 \rangle \rangle$$

$$\text{C-C-Swap2}\ \ D'\langle \tau_1, D\langle \tau_2, \tau_3 \rangle \rangle \equiv D\langle D'\langle \tau_1, \tau_2 \rangle, D'\langle \tau_1, \tau_3 \rangle \rangle$$

$$\text{C-C-Merge1}\ \ D\langle D\langle \tau_1, \tau_2 \rangle, \tau_3 \rangle \equiv D\langle \tau_1, \tau_3 \rangle$$

$$\text{C-C-Merge2}\ \ D\langle \tau_1, D\langle \tau_2, \tau_3 \rangle \rangle \equiv D\langle \tau_1, \tau_3 \rangle$$

Figure 1: Variational type equivalence.

choice types. For example, $A\langle A\langle \texttt{Int}, \texttt{Bool} \rangle, \texttt{Char} \rangle$ is the same as $A\langle \texttt{Int}, \texttt{Char} \rangle$ because the `Bool` alternative can never be selected.

The most important component of variational type inference is variational unification, which is unification modulo the type equivalence rules shown in Figure 1. One major challenge in solving the unification problem is to find a normal form for types that allows the comparisons of types, see [3] for details.

The error type $\bot$ was later introduced to make the variational type system more robust [2], because in the original system, if one program variant contains a type error, type inference fails for the whole program family. The addition of an error type allows the inference of *partial types* in which some alternatives may contain $\bot$, which means that program alternatives selected by decisions that lead in the variational type to $\bot$ contain a type error. The function `f` and its type $\tau_{\texttt{f}}$ provide a simple example. The addition of the error type creates another challenge to the unification, because we not only have to compute the most general unifier, but also the unifier that will result in the least number of type errors possible. Consider for example the following unification problem.

$$A\langle \texttt{Int}, \texttt{Bool} \rangle \equiv^{?} B\langle \texttt{Int}, \alpha \rangle$$

We can observe that there is one unavoidable type error because the second alternative of choice A is in conflict with the first alternative of choice B. Yet, the question is what should α be mapped to in order to make the result both most general and least problematic? We have proved that the variational unification problem is decidable and unitary [1, 3]. We also developed a unification algorithm that produces most general unifiers. Moreover, in presence of $\bot$, they also lead to minimum number of type errors. For the unification problem shown above, $\{\alpha \mapsto B\langle \beta, A\langle \texttt{Int}, \texttt{Bool} \rangle \rangle\}$ is such a unifier, where β is a fresh type variable.

3. A Type System for Lazy Typing

This section presents a type system for lambda calculus, extended by `let` bindings and data types. The type system realizes the idea of lazy typing by maintaining an additional environment with choice types that are analyzed after the typing is completed.

3.1 Syntax

The syntax of the expressions, types, and environments is shown in Figure 2. We use a bar to denote a sequence of elements, for example, $\overline{\phi}$ stands for $\phi_1, \ldots, \phi_n$.

Most of the definitions are as in other versions of lambda calculus, except for the addition of variational types, which introduce choices and error types. We could treat the conditional as a special case of case expressions, but the simpler syntactical structure simplifies some of the following discussions. As usual, the type environment Γ maps term variables to type schemas and is implemented as a stack. We use θ to denote unifiers, which are a subset

Term variables	x, y, z	Data constructors	C
Type variables	α, β	Type constructors	T
		Dimension names	A, B, D

Expressions	e	::=	$C\ \overline{e} \mid x \mid \lambda x.e \mid e\ e \mid \texttt{let}\ x\ \texttt{=}\ e\ \texttt{in}\ e$
		$\mid$	$\texttt{case}\ e\ \texttt{of}\ \overline{p\ \texttt{->}\ e} \mid \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e$
Patterns	p	::=	$C\ \overline{x}$

Monotypes	τ	::=	$\alpha \mid \tau \to \tau \mid T\ \overline{\tau}$
Variational types	ϕ	::=	$\tau \mid \bot \mid D\langle \overline{\phi} \rangle \mid \phi \to \phi$
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	::=	$\varnothing \mid \eta, \alpha \mapsto \phi$
Delayed typings	Δ	::=	$\varnothing \mid \Delta, D\langle \overline{\phi} \rangle$

Figure 2: Syntax of Expressions, Types, and Environments

of the type substitutions. We use Δ to gather choice types that are generated during the typing process. These choice types represent type errors whose processing is delayed until the whole expression is typed.

Finally, some auxiliary notation. We write $\eta_{/S}$ for $\{\alpha \mapsto \phi \in \eta \mid\ \alpha \notin S\}$. We also stipulate the conventional definition of FV, which computes the set of free type variables for a type, a type environment, and a substitution.

The type substitution $\eta(\sigma)$ substitutes the free type variables in σ with the corresponding images in η is defined as follows.

$$\eta(\bot) = \bot \qquad \eta(T\ \overline{\tau}) = T\ \overline{\eta(\tau)} \qquad \eta(D\langle \overline{\phi} \rangle) = D\langle \overline{\eta(\phi)} \rangle$$

$$\eta(\phi_1 \to \phi_2) = \eta(\phi_1) \to \eta(\phi_2) \qquad \eta(\forall \overline{\alpha}.\phi) = \forall \overline{\alpha}.\eta_{/\overline{\alpha}}(\phi)$$

$$\eta(\alpha) = \begin{cases} \alpha & \text{if } \alpha \notin dom(\eta) \\ \phi & \text{if } \alpha \mapsto \phi \in \eta \end{cases}$$

Note that we only define universally quantified variational types, and not variational polymorphic types since we can always transform the latter ones to the former ones. For instance, $D\langle \forall \alpha.\phi_1, \forall \beta.\phi_2 \rangle$ can be transformed to $\forall \alpha_1 \beta_1.D\langle \phi_1', \phi_2' \rangle$, where $\alpha_1 \notin FV(\phi_1)$ and $\beta_1 \notin FV(\phi_2)$, and $\phi_1' = \{\alpha \mapsto \alpha_1\}(\phi_1)$ and $\phi_2' = \{\beta \mapsto \beta_1\}(\phi_2)$.

3.2 Typing Rules

Figure 3 presents the typing rules for the expressions and types shown in Figure 2. Except for the case alternatives, the typing has the judgment $\Gamma \vdash e : \phi | \Delta$, which has two unusual features: (A) the result type is a variational type, and (B) the environment Δ collects all generated choice types. The latter facilitates the delay of computing type equivalence among case alternatives and conditional

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

VAR
$$\frac{\Gamma(x) = \forall \overline{\alpha}.\phi}{\Gamma \vdash x : \{\overline{\alpha \mapsto \phi'}\}(\phi) | \varnothing}$$

CON
$$\frac{\Gamma(C) = \forall \overline{\alpha}.\overline{\tau} \to T\ \overline{\alpha}}{\Gamma \vdash C : \{\overline{\alpha \mapsto \tau'}\}(\overline{\tau} \to T\ \overline{\alpha}) | \varnothing}$$

UNBOUND-V
$$\frac{x \notin dom(\Gamma)}{\Gamma \vdash x : \bot | \varnothing}$$

UNBOUND-C
$$\frac{C \notin dom(\Gamma)}{\Gamma \vdash C : \bot | \varnothing}$$

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

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

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

IF
$$\frac{(\Gamma \vdash e_i : \phi_i | \Delta_i)^{i:1..3} \qquad \pi = \phi_1 \bowtie \texttt{Bool} \qquad D \text{ is fresh} \qquad \phi_{23} = D\langle \phi_2, \phi_3 \rangle \qquad \phi = \pi \lhd \phi_{23}}{\Gamma \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : \phi | \Delta_1 \cup \Delta_2 \cup \Delta_3 \cup \{\phi_{23}\}}$$

CASE
$$\frac{\Gamma \vdash e : \phi | \Delta \qquad \Gamma \vdash p_i\ \texttt{->}\ e_i : \tau_i \to \phi_i | \Delta_i \qquad \pi_i = \phi \bowtie \tau_i \qquad \phi_i' = \pi_i \lhd \phi_i \qquad D \text{ is fresh} \qquad \phi' = D\langle \overline{\phi_i'} \rangle}{\Gamma \vdash \texttt{case}\ e\ \texttt{of}\ \overline{p_i\ \texttt{->}\ e_i} : \phi' | \bigcup_i \Delta_i \cup \Delta \cup \{\phi'\}}$$

$$\boxed{\Gamma \vdash p\ \texttt{->}\ e : \tau \to \phi | \Delta}$$

PAT
$$\frac{\Gamma(C) = \forall \overline{\alpha}.\overline{\tau} \to T\ \overline{\alpha} \qquad \overline{\alpha} \cap (FV(\Gamma) \cup FV(\overline{\tau}) \cup FV(\phi)) = \varnothing \qquad \eta = \{\overline{\alpha \mapsto \tau_p}\} \qquad \Gamma \cup \{\overline{x \mapsto \eta(\tau)}\} \vdash e : \phi | \Delta}{\Gamma \vdash C\ \overline{x}\ \texttt{->}\ e : T\ \overline{\tau_p} \to \phi | \Delta}$$

Figure 3: Lazy Typing Rules

branches, and consequently, only the rules IF and CASE produce new choice types; all other rules simply thread through or accumulate choice types from their premises.

Rules VAR, CON, ABS, and LET are all simply conservative extensions (that is, adding the threading of Δ) of the well-known rules. The rules UNBOUND-V and UNBOUND-C generate type errors for unbound variables and constructors, but such type errors are explicitly represented and threaded through the typing rules to support the lazy typing strategy.

Rule APP is different from the traditional rule and requires some explanation. The first two premises type function (e_1) and argument (e_2) independently of one another. The remaining three premises account, in a structured way, for the two possible ways in which the rule can fail and type errors can occur. First, the constraint that e_1's type ϕ_1 must be a function type is expressed in the third premise that calls an auxiliary function $\uparrow$, which tries to transform ϕ_1 into a function type. It is defined as follows (see [2] for more details).

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

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

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

$$\begin{aligned}
\phi \bowtie \phi &= \top & \phi_1 \to \phi_2 \bowtie \phi_1' \to \phi_2' &= (\phi_1 \bowtie \phi_1') \otimes (\phi_2 \bowtie \phi_2') \\
\bot \bowtie \phi &= \bot & D\langle \overline{\phi_1} \rangle \bowtie D\langle \overline{\phi_2} \rangle &= D\langle \overline{\phi_1 \bowtie \phi_2} \rangle \\
\phi \bowtie \bot &= \bot & D\langle \overline{\phi} \rangle \bowtie \phi' &= D\langle \overline{\phi \bowtie \phi'} \rangle \\
\phi \bowtie \phi' &= \bot & D\langle \phi' \rangle \bowtie \overline{\phi} &= D\langle \overline{\phi' \bowtie \phi} \rangle \\
& & T\ \overline{\tau_1} \bowtie T\ \overline{\tau_2} &= T\ \overline{\tau_1 \bowtie \tau_2}
\end{aligned}$$

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

$$\bot \lhd \phi = \bot \qquad \top \lhd \phi = \phi \qquad D\langle \overline{\pi} \rangle \lhd \phi = D\langle \overline{\pi \lhd \phi} \rangle$$

Figure 4: Typing patterns, matching, and masking

For example, $\uparrow(A\langle \texttt{Int} \to \texttt{Bool}, \texttt{Bool} \to \texttt{Int} \rangle)$ lifts the function type out of the choice type and yields $A\langle \texttt{Int}, \texttt{Bool} \rangle \to A\langle \texttt{Bool}, \texttt{Int} \rangle$, while $\uparrow(A\langle \texttt{Int} \to \texttt{Bool}, \texttt{Int} \rangle)$ can succeed only by introducing an error type: $A\langle \texttt{Int}, \bot \rangle \to A\langle \texttt{Bool}, \bot \rangle$.

The second constraint is that the type of e_2 matches the argument type of ϕ_1, which is expressed in the fourth premise. Matching two types results in a *typing pattern* (defined in Figure 4) that captures the common structure of the two types matched and also which parts of the structure match (represented by $\top$) and which don't (represented by a type error $\bot$).

Matching plain types is obvious and always results either in $\top$ or $\bot$. For example, $\texttt{Int} \bowtie \texttt{Int} = \top$ and $\texttt{Int} \bowtie \texttt{Bool} = \bot$. Matching is more interesting for variational types that contain choices, because it produces partial matches, which provide a more detailed view on where type errors occur. For example, $A\langle \texttt{Int}, \texttt{Bool} \rangle \bowtie \texttt{Int} = A\langle \top, \bot \rangle$. When matching two function types, we first match the corresponding argument types and result types. Based on the returned typing patterns, we use $\otimes$ to build a new typing pattern each of whose alternative is $\top$ if and only if that alternative in both typing patterns is $\top$. Consider, for example, $\texttt{Int} \to A\langle \texttt{Bool}, \texttt{Int} \rangle \bowtie B\langle \texttt{Int}, \bot \rangle \to \texttt{Bool}$. Matching the arguments produces $B\langle \top, \bot \rangle$, and matching the return types produces $A\langle \top, \bot \rangle$. Thus the final result for matching the two types is $A\langle B\langle \top, \bot \rangle, \bot \rangle$, because only the first alternative in both A and B is $\top$. Matching is defined in Figure 4. The definition contains overlapping cases and assumes that the most specific rules that match are applied first (see [2]).

In the third step, we mask the return type of e_1, if which is a function type over the typing pattern we computed before. Essentially, this step makes the return type to go through for the alternatives where argument type matches the type of the argument and makes it blocked, thus get the type $\bot$ for other situations. The masking operation is defined in Figure 4, which replaces all the occurrences of $\top$ by ϕ, the type under the masking operation.

While rule APP can introduce type errors, it will not itself produce choice types. This happens in the rules IF and CASE. Let's consider rule IF first. After independently typing all three argument expressions, a typing pattern is created for the condition, which must be of type `Bool`. Then we create a choice type (ϕ_{23}) for the types of the two branches, which is added to the Δ environment as a delayed typing decision. It is also returned as the type of the conditional if the condition is of type `Bool` (this constrained is expressed through the masking with the computed typing pattern π). As an example consider the following expression e_1.

$$e_1 = \texttt{let f = if True then odd else not in 3}$$

The application of the IF rule yields `Bool` for ϕ_1, which means matching returns the typing pattern $\top$. Further we obtain $\phi_2 = \texttt{Int} \to \texttt{Bool}$ for the `then` branch and $\phi_3 = \texttt{Bool} \to$

`Bool` for the `else` branch. The generated choice type is thus $D\langle \texttt{Int} \to \texttt{Bool}, \texttt{Bool} \to \texttt{Bool}\rangle$, which will be placed into the Δ environment. Since masking with the typing pattern $\top$ has no effect on this type, the return type ϕ is the same. Now applying the rule LET we obtain as a result for e_1 the following type plus delayed typing decision.

$$\texttt{Int} | \{D\langle \texttt{Int} \to \texttt{Bool}, \texttt{Bool} \to \texttt{Bool}\rangle\}$$

Since $D\langle \texttt{Int} \to \texttt{Bool}, \texttt{Bool} \to \texttt{Bool}\rangle$ can't be simplified to a monotype, this expression is not well-typed. The type simplification is discussed shortly in Section 3.3.

The CASE rule works in a similar way. First, a function type $\tau_i \to \phi_i$ is derived for each case, and a corresponding typing pattern π_i is computed by matching the type of the pattern τ_i against the type ϕ of the scrutinized expression e. With this typing pattern we mask the return type ϕ_i of each case and gather all return types in a choice type ϕ' that is the result type for the case expression, which is also placed into the Δ environment for later analysis.

Finally, the PAT rule describes how to infer the function type for patterns that are used in the rule CASE. The type is obtained by extending the type environment Γ with an assumption for each pattern variable that is taken from the argument type of the constructor C. With this extended environment the expression on the RHS of the pattern is checked. This is a fairly standard rule, and there are no interesting aspects from the perspective of lazy typing.

3.3 Correctness of Lazy Typing

Since lazy typing produces results in more cases than traditional type systems do, the question is whether (A) it produces the same results for type-correct programs, and (B) whether in all other cases the result is interpreted as a type error. Taken together, (A) and (B) express that lazy typing is a conservative extension of traditional type systems. In this section we will demonstrate that this is indeed the case. This is done in several steps.

The first observation is that the lazy type system will always produce a result.

LEMMA 1. *For any given e and Γ, there exist a type ϕ and a delayed typing environment Δ such that $\Gamma \vdash e : \phi | \Delta$.*

PROOF SKETCH. The proof is based on the fact that for any expression one of the typing rules in Figure 3 is applicable and has, by induction, all of its premises succeed. □

Next, we show that the lazy typing system is a conservative extension of the Hindley-Milner type system. We prove this by showing that the two type systems produce the same result for well-typed programs. But before we can formally talk about this property, we have to first define how to characterize what well typing means in the lazy typing system. We provide the necessary definitions in Figure 5. The judgment $\Gamma \vdash_\Delta e : \tau$ says that e is well typed and has the type τ under lazy typing, which is the case when the variational return type as well as all delayed typing decisions in Δ (if any) can be reduced to monotypes.

The reduction of types is defined as the reflective, transitive closure of the rewrite relationship $\rightsquigarrow$ defined in Figure 5, which simplifies type expressions. Simplification reduces function types that contain type errors to type errors, replaces choices whose alternatives are all equal (but not a type error) with one of these alternatives, and applies choice domination by replacing nested choices in the same dimension with the corresponding nested alternative. Otherwise, simplification descends recursively into function and choice types and pushes function types down into choice types. The simplification rule for choice dominance makes use of the selection operation $\lfloor \phi \rfloor_{D.i}$, also defined in Figure 5, which replaces each occurrence of a D choice with its ith alternative.

We observe that the simplification relationship is confluent.

$$\boxed{\Gamma \vdash_\Delta e : \tau} \qquad \frac{\Gamma \vdash e : \phi | \Delta \qquad \phi \rightsquigarrow^* \tau \qquad \Delta \Downarrow}{\Gamma \vdash_\Delta e : \tau}$$

$$\boxed{\Delta \Downarrow} \qquad \frac{}{\varnothing \Downarrow} \qquad \frac{\Delta \Downarrow \qquad D\langle \overline{\phi} \rangle \rightsquigarrow^* \tau}{\Delta, D\langle \overline{\phi} \rangle \Downarrow}$$

$$\boxed{\phi \rightsquigarrow \phi}$$

$$\frac{\phi_l \rightsquigarrow \phi_l'}{\phi_l \to \phi_r \rightsquigarrow \phi_l' \to \phi_r} \qquad \frac{\phi_r \rightsquigarrow \phi_r'}{\phi_l \to \phi_r \rightsquigarrow \phi_l \to \phi_r'}$$

$$\frac{\phi_i \rightsquigarrow \phi_i'}{D\langle \overline{\phi} \rangle \rightsquigarrow D\langle [\phi_i'/\phi_i]\overline{\phi} \rangle} \qquad \frac{\lfloor \phi_i \rfloor_{D.i} = \phi_i' \qquad \phi_i \neq \phi_i'}{D\langle \overline{\phi} \rangle \rightsquigarrow D\langle [\phi_i'/\phi_i]\overline{\phi} \rangle}$$

$$\bot \to \phi \rightsquigarrow \bot \qquad \phi \to \bot \rightsquigarrow \bot \qquad \frac{\phi \neq \bot}{D\langle \phi, \ldots, \phi \rangle \rightsquigarrow \phi}$$

$$\phi_1 \to D\langle \overline{\phi} \rangle \rightsquigarrow D\langle \overline{\phi_1 \to \phi} \rangle \qquad D\langle \overline{\phi} \rangle \to \phi_1 \rightsquigarrow D\langle \overline{\phi \to \phi_1} \rangle$$

$$D_1\langle \overline{\phi_1} \rangle \to D_2\langle \overline{\phi_2} \rangle \rightsquigarrow D_1\langle \overline{\phi_1 \to D_2\langle \phi_2 \rangle} \rangle$$

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

$$\lfloor \tau \rfloor_s = \tau \qquad \lfloor \phi_1 \to \phi_2 \rfloor_s = \lfloor \phi_1 \rfloor_s \to \lfloor \phi_2 \rfloor_s$$
$$\lfloor \bot \rfloor_s = \bot \qquad \lfloor B\langle \overline{\phi} \rangle \rfloor_{A.i} = B\langle \overline{\lfloor \phi \rfloor_{A.i}} \rangle \quad \text{if } A \neq B$$
$$\lfloor B\langle \overline{\phi} \rangle \rfloor_{B.i} = \phi_i$$

Figure 5: Well typing, simplification, and selection

LEMMA 2 ($\rightsquigarrow$ is confluent). *For any ϕ, if $\phi \rightsquigarrow^* \phi_1$ and $\phi \rightsquigarrow^* \phi_2$, then there exists some ϕ' such that $\phi_1 \rightsquigarrow^* \phi'$ and $\phi_2 \rightsquigarrow^* \phi'$.*

PROOF SKETCH. The proof is based on the fact that $\rightsquigarrow$ is both locally confluent and terminating. First, the relation is locally confluent because when simplifying a function type, the ordering of simplifying the argument and the return type does not matter. The same is true for simplifying choice types, that is, which alternative is simplified first doesn't matter. For pushing function types down over choice types, the ordering is specified in the rewriting relations. Second, the relation is also terminating because no rule is applicable when there are no nested choices in the same dimension and when function return and argument types are monotypes, and the rules make steady progress toward that situation. □

Now we can relate our type system to the Hindley-Milner type system, which is represented by the judgment $\Gamma \vdash e : \tau$.

THEOREM 1 (Correctness of Lazy Typing). *For any given e and Γ, $\Gamma \vdash_\Delta e : \tau \iff \Gamma \vdash e : \tau$.*

PROOF. First, we show $\Gamma \vdash e : \tau$ implies $\Gamma \vdash_\Delta e : \tau$. We observe that when $\Gamma \vdash e : \tau$ holds, there are no unbound variables or constructors in e, that is, the rules UNBOUND-V and UNBOUND-C will not be applicable, and no $\bot$ will be introduced. Since both type systems are syntax-directed, we can build a proof based on an induction over typing derivations that shows that the two type systems behave in the same way. Specifically, we can observe that any additional premises in the lazy typing system are not applicable in the absence of type errors.

We only show here the case for function applications. The construction for other rules is quite similar. First, we need the following two auxiliary lemmas.

LEMMA 3 ($\uparrow$ preserves $\rightsquigarrow$). *If* $\phi \rightsquigarrow^* \phi_1 \to \phi_2$, *then* $\uparrow(\phi) = \phi_3 \to \phi_4$ *with* $\phi_3 \rightsquigarrow^* \phi_1$ *and* $\phi_4 \rightsquigarrow^* \phi_2$.

LEMMA 4 ($\rightsquigarrow$ preserves $\bowtie$). *If* $\phi_1 \rightsquigarrow \phi_2$, *then* $\phi_1 \bowtie \phi_3 = \phi_2 \bowtie \phi_3$. *In particular, if* $\phi_1 \rightsquigarrow^* \phi_2$ *and* $\phi_3 \rightsquigarrow^* \phi_4$, *then* $\phi_1 \bowtie \phi_3 = \phi_2 \bowtie \phi_4$.

Returning to the proof, from $\Gamma \vdash e_1\ e_2 : \tau$ we know that the premises $\Gamma \vdash e_1 : \tau_l \to \tau$ and $\Gamma \vdash e_2 : \tau_l$ hold. Based on the induction hypothesis, we know $\Gamma \vdash_\Delta e_1 : \tau_l \to \tau$ and $\Gamma \vdash_\Delta e_2 : \tau_l$, which translates into $\Gamma \vdash e_1 : \phi_1|\Delta_1$ where $\phi_1 \rightsquigarrow^* \tau_l \to \tau$ and $\Delta_1 \Downarrow$, and $\Gamma \vdash e_2 : \phi_2|\Delta_2$ where $\phi_2 \rightsquigarrow^* \tau_l$ and $\Delta_2 \Downarrow$. Based on Lemma 3, there exist ϕ_{ll} and ϕ_{lr} such that $\phi_{ll} \to \phi_{lr} = \uparrow(\phi_1)$ and $\phi_{ll} \rightsquigarrow^* \tau_l$ and $\phi_{lr} \rightsquigarrow^* \tau$. Based on Lemma 4, $\phi_{ll} \bowtie \phi_2 = \tau_l \bowtie \tau_l = \top$. Moreover, $\top \lhd \phi_{lr} = \phi_{lr}$. Thus following the rule APP, $\Gamma \vdash e_1\ e_2 : \phi_{lr}|\Delta_1 \cup \Delta_2$. Since $\Delta_1 \Downarrow$ and $\Delta_2 \Downarrow$, it follows $\Delta_1 \cup \Delta_2 \Downarrow$. Also, we have seen that $\phi_{lr} \rightsquigarrow^* \tau$, proving that $\Gamma \vdash_\Delta e_1\ e_2 : \tau$.

The proof for the other direction is performed similarly. □

From this theorem it follows directly that lazy typing does neither produce false positive or false negative type errors, which can be summarized in the following two corollaries. We use the notation $\Delta \not\Downarrow$ to express that Δ, in case it is not empty, cannot be reduced to a monotype.

COROLLARY 1. *Given e and* Γ, *if* $\Gamma \vdash e : \phi|\Delta$ *and there is no* τ *such that* $\phi \rightsquigarrow^* \tau$ *or* $\Delta \not\Downarrow$, *then there is no* τ' *such that* $\Gamma \vdash e : \tau'$.

On the other hand, if an expression is not typeable in the Hindley-Milner system, then our type system will report a type error for that expression as well.

COROLLARY 2. *Given e and* Γ, *if there is no type* τ *such that* $\Gamma \vdash e : \tau$, *then* $\Gamma \vdash e : \phi|\Delta$ *implies that there is no* τ' *such that* $\phi \rightsquigarrow^* \tau'$ *or* $\Delta \not\Downarrow$.

The type system is implemented in two steps. The first step is a variant of algorithm $\mathcal{W}$ [7] that returns a variational type, a unifier that maps type variables to variational types, and an environment that contains all the delayed constraints. The second step consists of a post-unifying process, in which we try to resolve all variational types by unifying all the alternatives in each choice type. The technical details can be found in the Appendix A.

4. Localizing and Reporting Type Errors

The lazy typing system produces a variational type ϕ and an environment Δ containing delayed decisions represented as choice types. This information has yet to be turned into a type error message. In this section we describe a method that exploits the type information to derive an as precise as possible pointer to the location of the type error in the source code. Moreover, in some cases we will also derive a suggestions for a new type for that expression.

Our approach to reporting type errors and suggestions is conservative in the sense that we only report in the cases that we can track the type error down to a single expression. In practical terms this means, if the expression is a case alternative or a branch of a conditional, then removing that expression will fix the corresponding type error. Moreover, when we also present an expected type for that expression, then the type error will be really fixed if the expression is changed to the reported expected type.

To map types to their originating expressions, we employ a mapping γ that maps from dimension names (that occur in ϕ or Δ) or selectors to nodes in the abstract syntax tree whose expression caused the generation of the choice type or the corresponding alternative. Such a mapping can be constructed as follows. Whenever we generate a fresh choice type (in rule IF or CASE), we can actually produce a pair consisting of the choice type and an expression. For example, in the rule IF, we create the pairs (ϕ_2, e_2) and (ϕ_3, e_3) to establish the links from the selectors $D.1$ and $D.2$ to the respective expressions e_2 and e_3. We also create the pair $(D\langle\phi_2,\phi_3\rangle, \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3)$ to establish the link from the choice D to the conditional expression. In a similar way, we create pairs (ϕ'_i, e_i) and $(\phi', \texttt{case}\ e\ \texttt{of}\ \overline{p_i\ \texttt{->}\ e_i})$ to track the selectors and choice type in the CASE rule.

We also track applied expressions in rule APP whenever ϕ (which results from masking with the typing pattern π) results in type $D\langle\overline{\bot}\rangle$ or $\bot$, because either of these types indicates that the applied function does not work under any circumstance and is thus the source of a type error. We record this information in a separate mapping μ by adding a pair $(D\langle\overline{\bot}\rangle, e_1)$ (or $(\bot, e_1)$) in such situations. Note that we create at most one such entry for any path along the left spine from the leaf of an expression to the root to locate the most nested violating function application only.

From the previous section (specifically, Corollary 2) we know that a type error occurs only if ϕ cannot be reduced to a monotype or if Δ is not empty and some of its choice types cannot be reduced to monotypes. Thus, we analyze these two pieces of type information to find out about the source of the error. We describe how the algorithm works on a high level, illustrating it with examples as we go along. Some of the examples are taken from the table in Figure 6 and will be discussed in more depth after the algorithm has been presented.

(1) Type Normalization
First, we normalize ϕ and the types in Δ using the simplification $\rightsquigarrow$ defined in Figure 5. This will eliminate gratuitous choices such as $A\langle\texttt{Int},\texttt{Int}\rangle$ and allows us to consider fewer patterns in the analysis.

For the `fib` example, we first simplify the result type $\texttt{Int} \to B\langle A\langle\bot,\texttt{Int}\rangle, A\langle\bot,\texttt{Int}\rangle,\texttt{Int}\rangle$ to

$$\phi = B\langle A\langle\bot, \texttt{Int} \to \texttt{Int}\rangle, A\langle\bot, \texttt{Int} \to \texttt{Int}\rangle, \texttt{Int} \to \texttt{Int}\rangle$$

The delayed decisions represented as choice types are already in simplified form.

$$\Delta = \{A\langle\texttt{[Int]},\texttt{Int}\rangle, B\langle A\langle\bot,\texttt{Int}\rangle, A\langle\bot,\texttt{Int}\rangle,\texttt{Int}\rangle\}$$

The next step localizes errors and (potentially) creates suggestions. It consists of two parts. Step (2.1) determines whether or not the error is contained in an alternative of a choice type, and step (2.2) actually performs error localization in nested types. Given ϕ and $\Delta = \{\phi_1, \ldots, \phi_n\}$, steps (2.1) and (2.2) are iterated over all types ϕ, ϕ_1,...,ϕ_n, while the results are accumulated.

(2.1) Stop or Refine
Next we have to determine whether the inferred type ϕ indicates a type error and whether it is located where choice types have been generated or where choice types (or $\bot$) are used. Specifically, we first check whether $\phi \rightsquigarrow \tau$, in which case there is no occurrence of a type error at this point and we can stop. (There might be errors in Δ, and these will be processed in different iterations of this step.) If $\phi \rightsquigarrow \bot$, there is a type error at position $\mu(\phi)$. Finally, if $\phi \rightsquigarrow D\langle\ldots\rangle$, we have to determine if $e' = \gamma(\phi)$, the source of the choice type, or its use $\mu(\phi)$ contains a type error, in which case we can't determine a more refined location in, say, one of the alternatives $\gamma(D.i)$. Specifically, e' contains a type error if it satisfies any one of the following three conditions. From now on we assume $\phi \rightsquigarrow D\langle\phi_1, \ldots, \phi_n\rangle$, and we write more shortly $D \in \Delta$ if there is a choice type $D\langle\ldots\rangle \in \Delta$.

(i) When $\exists i, j : \phi_i = \tau \neq \tau' = \phi_j$. In this case there is not enough contextual information to further refine the type error to $\gamma(D.i)$ for a specific alternative. Case (b) in Figure 6 provides a simple example.
(ii) When $\forall i : \phi_i = \bot$ and $D \in \Delta$. In this case, we know that all the alternatives or branches in $\gamma(D)$ have different types. Case (g) in Figure 6 is such an example.

(iii) When $\exists i,j : \phi_i = B\langle\overline{\phi_B}\rangle \neq B\langle\overline{\phi'_B}\rangle = \phi_j$. In this case, $\gamma(D.i)$ and $\gamma(D.j)$ have different types, and there is no indication about which of the two should be the correct type. Case (i) in Figure 6 provides such an example where we report $\gamma(A)$ as a source of some type error.

When all the monotypes of $\phi_1,\ldots,\phi_n$ are identical, we report $\gamma(D.i)$ as an error source if $\phi_i = \bot$. Case (c) and (h) in Figure 6 are such examples. In all other cases, neither $\gamma(D)$ nor $\gamma(D.i)$ can be said to contain type error. Cases (d), (j), (k) and (l) in Figure 6 are such examples.

In the case of `fib`, none of the conditions applies. We therefore do not report a type error for $\gamma(B)$.

The use of the expression with type $D\langle\overline{\bot}\rangle$ can also be the source of the type error, in which case the algorithm also has to stop. This is the case when $\forall i : \phi_i = \bot$ and $D \notin \Delta$. We report $\mu(\phi)$ as the error location.

(2.2) Reporting Nested Type Errors

Next, we have to decide how to report type errors for the choice types nested in D. If all the alternatives that do not contain $\bot$ are identical, then we will suggest that type alternative as a replacement for all $\bot$-containing types. This works because we can simplify the choice type into a monotype after that replacement. For example, $D\langle\texttt{Int},\bot,\texttt{Int}\rangle$ can be transformed to `Int` if we replace $\bot$ with `Int`. On the other hand, this does not work for $D\langle\texttt{Int},\bot,\texttt{Bool}\rangle$. In the case of `fib` we find exactly this is the situation since all non-$\bot$ alternatives are given by the type $\texttt{Int}\to\texttt{Int}$. If we find different non-$\bot$ alternatives, we cannot suggest a type since we have no basis to pick one of the alternatives.

More specifically, we have to perform the following two steps.

(1) Let p be a sequence of selectors with which we can extract the common non-$\bot$ type from ϕ. In the `fib` example, we find $p = [B.1, A.2]$. For each type ϕ_i (that is, $\lfloor\phi\rfloor_{D.i}$), do the following.

(2) (a) If $\phi_i = \bot$, report $\gamma(D.i)$ as the source of type error. In the `fib` example, no $\bot$ is directly nested in B, so there is no $\gamma(B.i)$ reported as a type error.

(b) If ϕ_i is a monotype, we don't have to do anything since the type does not contain a type error.

(c) If ϕ_i is itself another choice type, say $D'\langle\ldots\rangle$, we recursively apply the current method. In the `fib` example, this leads to recursively visiting $A\langle\bot,\texttt{Int}\to\texttt{Int}\rangle$. If we repeat steps (1) and (2) on $A\langle\bot,\texttt{Int}\to\texttt{Int}\rangle$, we find $p = [A.2]$. For step (2), we encounter $\bot$ when we visit the first alternative of A, which means case (2a) applies, and we report $\gamma(A.1)$, that is, `[0]` as a type error and $\Delta(p) = \texttt{Int}$ as the suggested type.

(3) Conflict Resolution

By analyzing the result type ϕ of e, we report all the potential errors of the expressions used by e, which is a reflection of our key idea: We detect type errors for expressions by analyzing how they are used. One question that remains is how we can aggregate the type information resulting from the potentially many uses of an erroneous expression, and what conclusion about the type error in that expression can be drawn? Suppose e is the expression which generated $D\langle\ldots\rangle$, and e_i is the subexpression in e that generated $D.i$. We can distinguish between the following different cases.

(1) If all the uses suggest that e is the source of some type error, then we report e as the source of the type error.

(2) If some use indicates that e_i is the source of a type error, and if the suggestion is that e_i have the same type as e_j, then we report that e_i is erroneous and that it should be changed so that it has the same type as e_j.

	ϕ	#Errors	Location(s)	Suggestion
(a)	$\tau^{\clubsuit}$	0		
(b)	$D\langle\tau_1,\tau_2\rangle^{\diamondsuit}$	1	$\gamma(D)$	*none*
(c)	$D\langle\tau,\bot\rangle$	1	$\gamma(D.2)$	$\Delta(D.1)$
(d)	$B\langle A\langle\bot,\tau\rangle,\tau\rangle$	1	$\gamma(A.1)$	$\Delta(A.2)$
(e)	$A\langle\tau_1,B\langle\bot,\tau_2\rangle\rangle$	1	$\gamma(B)$	*none*
(f)	$D\langle\bot,\bot\rangle^{\heartsuit}$	1	$\mu(D)$	*none*
(g)	$D\langle\bot,\bot\rangle^{\spadesuit}$	1	$\gamma(D)$	*none*
(h)	$A\langle\bot,B\langle\tau,\bot\rangle\rangle$	2	$\gamma(B.2)$ $\gamma(A.1)$	$\Delta(B.1)$ $\Delta([A.2,B.1])$
(i)	$A\langle B\langle\tau,\bot\rangle,$ $B\langle\bot,\tau\rangle\rangle$	2	$\gamma(B)$ $\gamma(A)$	*none*
(j)	$A\langle B\langle\tau,\bot\rangle,$ $D\langle\bot,\tau\rangle\rangle$	2	$\gamma(B.2)$ $\gamma(D.1)$	$\Delta(B.1)$ $\Delta(D.2)$
(k)	$A\langle B\langle\tau_1,\bot\rangle,$ $D\langle\bot,\tau_2\rangle\rangle$	2	$\gamma(B)$ $\gamma(D)$	*none*
(l)	$A\langle B\langle\bot,$ $D\langle\tau,\bot\rangle\rangle,$ $E\langle\bot,\tau\rangle\rangle$	3	$\gamma(B.1)$ $\gamma(D.2)$ $\gamma(E.1)$	$\Delta([B.2,D.1])$ $\Delta(D.1)$ $\Delta(E.2)$

$^{\clubsuit}\Delta = \varnothing \quad ^{\diamondsuit}\Delta \neq \varnothing \quad ^{\heartsuit}D \notin \Delta \quad ^{\spadesuit}D \in \Delta$

Figure 6: Examples of Result Types and Reported Error Locations

(3) If some use suggests that e_i is the source of some type and it should have the type of expression e_j, and at the same time, some other use suggests that e_j contains a type error and should have the same type of e_i, the context reveals a conflict of preference about the type for e. Thus, we report e as the source of the type error. Case (i) in Figure 6 is such an example. For $B\langle\tau,\bot\rangle$, $\gamma(B.2)$ is suggested to have type $\Delta(B.1)$ and for $B\langle\bot,\tau\rangle$, $\gamma(B.1)$ is suggested to have the type $\Delta(B.2)$. As a result, $\gamma(B)$ is the source of the type error.

In the remainder of this section we will first illustrate the algorithm on several (small) example cases to illustrate how different nestings of choice types, $\bot$, and monotypes can be interpreted to localize type errors. We conclude with an argument for the correctness of the algorithm. The examples are collected in Figure 6, where we show the result types for expressions (ϕ), the number of type errors, the location of each type error and the expected type for each expression to correct that type error. To simplify the presentation, each choice has two alternatives only. In cases we can't suggest a type for the erroneous expression, we use *none* in the last column.

We assume that ϕ and Δ are most simplified. We also assume that $\tau_1 \neq \tau_2$. For cases (a), (b), (f) and (g) we have attached an additional condition on Δ. Finally, if ϕ is a choice type in Δ whose outermost choice is D, we define $\Delta(D) = \phi$ and $\Delta(D.i) = \lfloor\phi\rfloor_{D.i}$. Moreover, we extend selection to paths (that is, sequences of selectors) as follows: $\Delta([s_1,\ldots,s_n]) = \lfloor\cdots\lfloor\phi\rfloor_{s_1}\cdots\rfloor_{s_n}$.

In case (a), the expression is well typed, and thus there is nothing to report. In case (b), the result type is a choice with two different alternatives. We report $\gamma(D)$ as the source of the type error, because neither of the two alternatives is obviously better than the other, which is also why we cannot generate a suggestion.

Cases (c) and (d) are typical for expressions that have one type error. In both cases, we report the branch that corresponds to $\bot$ as the error location, and we suggest as the the correct type the corresponding other alternative. Case (d) is basically the same result as for the `fib`, except for the third alternative. Case (e) is more interesting. Here we find a type error at $\gamma(B)$. (Note that it can't be $\gamma(B.1)$ since changing $\gamma(B.1)$ to have the type of $\gamma(B.2)$ would *not* eliminate the type error in $\gamma(A)$.) There also must be a type error at $\gamma(A)$ since the two alternatives of the final type are different.

Similarly, we can't suggest a type since no single suggestion would eliminate that type error.

Cases (f) and (g) show that with the same type but different environments Δ, the reported error locations can be different. In case (f), the caller of $\gamma(D)$ very likely contains a type error because $\gamma(D)$ itself is type correct, witnessed by the fact that $D \notin \Delta$. In case (g), $\gamma(D)$ has a high probability of being type incorrect while there may or may not be a type error in $\mu(D)$.

In case (h), the suggested type for $\gamma(A.1)$ is the type $\Delta([A.2, B.1])$, which means to find in Δ the choice type ϕ_i with dimension A, and then make the selection $\lfloor \lfloor \phi_i \rfloor_{A.2} \rfloor_{B.1}$.

In case (i), the context yields conflicting information about the use of choice B, which is why we cannot provide a precise error location for an alternative of B and can produce no suggestion. There is also a type error in $\gamma(A)$ because changing either alternative of B will not fixed the error in A. For example, if $\gamma(B.1)$ gets the type $\Delta(B.2)$, the result type will change to $A\langle B\langle \bot, \bot \rangle, \tau \rangle$. On the other hand, suppose $\gamma(B.2)$ gets the type $\Delta(B.1)$, then the result type will be $A\langle \tau, B\langle \bot, \bot \rangle \rangle$. Although case (j) is quite similar to case (i), the error locations and suggested types differ significantly. It is clear that there are type errors in $\gamma(B.2)$ and $\gamma(D.1)$. However, we don't report that there is a type error in $\gamma(A)$ because the type error can be fixed by letting $\gamma(B.2)$ to have the type $\Delta(B.1)$ and letting $\gamma(D.1)$ to have the type $\Delta(D.2)$.

Case (k) again illustrates how a minor difference in the result type can lead to differences in error reporting. The error locations are the same as for the case (j), but we have no types to suggest because, whatever the type suggested, the type error in $\gamma(A)$ will persist. Finally, case (l) illustrates a slightly bigger example.

We claim that the reported error locations are correct in the following counterfactual sense:

> *If the reported expression is not changed, then some type error will not be removed, or more involved changes at other places will be required to remove that type error.*

Moreover, type-change suggestions are correct in the following sense.

> *At least one type error will be removed if the blamed expression will be changed to have the suggested type.*

We will not formalize these two points, but rather present some informal reasons for why we believe they are true.

First, whenever a conditional or a case expression is blamed to have a type error, we do this only when one of the three conditions in step (2.1) is satisfied. Any of these conditions implies that at least two branches or case alternatives have different types. To fix the type error, the conditional or the case expression must changed.

Second, when the use of a function that has multiple branches, all with the same result type, leads to a type error, we report the use of the function as the source of the type error. This is because if we assumed the use to be correct, we would have to change the type for the function, which very likely would lead to changes in all branch of the function, a larger change.

Third, consider the case when the reported error location is e_i whose type is τ_i and which is a branch in a larger expression e. According to the cases (2) and (3) in step (3), some context has favored the type of e_i to be changed to the type of e_j, say τ_j. If we make the suggested change, then according to the beginning of step (2.2), a type error will be eliminated. Now assume we don't make the suggested change and instead change the expression e_j to have type τ_i. In that case the context that works with e_j won't work anymore because e_j will have a type τ_i, which is does not work with that context. So we have to change that context to make it work with e_i, which can cause other potential cascading changes. At the same time, the change of e_j to have type τ_i will not work with some other context since otherwise that context would have suggested e_j to have type τ_i and a conflict would have been detected according to case (3) in step 3, which in turn would have caused the suggestion not to be made in the first place. Thus, if the located expression is not changed, other changes will cause more type errors. If that expression is not changed to the suggested type, the type error will not be fixed.

5. Evaluation

To evaluate the usefulness of our approach, we have selected a set of examples that were previously introduced in the relevant literature to illustrate the behavior of the different approaches and also reveal important challenges. Most of the programs that we have selected involve conditionals or case expressions since these are the constructs that demonstrate the effects of the lazy typing technique. (This is not a severe restriction since pattern matching and case expression are pervasive in functional programs.) As mentioned before, we do not consider lazy typing as a replacement for other techniques, but rather as an addition that could help improve other approaches. Therefore, our evaluation is focused on finding out how well lazy typing could contribute in those relevant cases. Since few examples introduced in the literature contain more than one type errors, we have also added a few example programs containing two type errors. All the programs are shown in the Appendix B.

We have implemented a Haskell prototype for the lazy typing system and ran the prototype on the set of examples. To compare the results with previous approaches, we also ran GHC, the de facto standard, and the change suggesting tools Helium and Seminal on the examples.

We have evaluated the output produced by each tool in two regards. First, we judged how accurate the reported location of the type error was and classified it as either *correct* when it was spot on, *relevant* if it was closely related but not quite exact, and *incorrect* when the error message was actually pointing at a wrong part of the code. Second, we judged the accuracy of type suggestions. Here there were basically only two possibilities, either the suggested type was *correct*, or the suggestion was *incorrect*. In some cases, when not enough information is available to reliably generate a type suggestions, lazy typing will withhold judgment and refrain from making a (potentially incorrect) suggestion. (Helium and Seminal always make a statement about an expected type and thus commit to a suggestion even in cases when not enough evidence is available. GHC makes suggestions sometimes.) We conservatively grouped this case of a *missing suggestion* together with the *incorrect* outcome.

Figure 7 shows the evaluation result for different tools and example programs. We represent the evaluation for a particular tool and example using an overlay of two more or less filled half circles. The left half of the circles represents the quality of reported error location, and the right half represents the presence and correctness of suggested types.

We can observe that lazy typing correctly reports error locations in every single program and only lacks type suggestions in four programs, for three of which suggestions shouldn't be produced anyway. Whenever a type is suggested by lazy typing, it is a correct suggestion. Only in the `map` example are two other tools (Helium and GHC) able to correctly suggest a type when lazy typing fails to do so. In all other cases, lazy typing performs just as well or better than all other approaches.

Why do Helium and GHC perform better than lazy typing in the `map` example? When the APP rule fails, lazy typing only records the error locations and dones't track unification failures. Thus when `Bool` fails to unify with `a -> b`, lazy typing is unable to exploit this information, unlike Helium and GHC that do this very well.

Example	Lazy Typing	Seminal	Helium	GHC
`fib`	●	○	○	○
`split`[27]	●	◕	○	○
`map`[27]	◐	◐	●	●
`add3`[12]	●	●	●	○
`insert`[23]	●	◕	●	○
`strlist`[15]	●	●	●	●
`strlist1`	●	○	○	○
`if1`[23]	◐	◴	◴	◴
`if2`	●	○	○	○
`plus`[15]	◐◐	◐◐	◐◐	◐○
`fibbool`	●●	◴○	●○	◴○
`condfun`	●●	◴○	◴○	○○
`fiblist`	◐◐	◴○	◴○	◴○

Error Location	Suggested Type
◐ correct	◑ correct
◴ relevant	
○ irrelevant	○ incorrect or absent

Figure 7: Evaluation Results for Different Tools

6. Related Work

The challenge of accurately reporting type errors and producing helpful error messages has received considerable attention in the research community. Improvements for type inference algorithms have been proposed that are based on changing the order of unification, suggesting program fixes, and using program slicing techniques to find all program locations involved in type errors. Lazy typing can pinpoint the most likely error locations and suggest expected types when enough contextual information is available. Thus our approach can be used to complement most of the previously proposed approaches. Heeren [10] and Yang et al. [29] have nicely summarized and discussed much of the older work in this area. We will therefore instead focus our discussion on comparisons on the technical level as well as on the work that was not covered by their summaries.

Methods that are based on changing the order in which unification problems are solved include the algorithm $\mathcal{M}$ [13], the algorithm $\mathcal{G}$ [14], the algorithm $\mathcal{W}^{SYM}$ [17], and the algorithm $\mathcal{IEI}$ [28]. Our proposed lazy typing is also a unification reordering approach in the sense that conditional branches and case alternatives are typed independently and their unification problems are solved at the end of type inference process. A major difference, however, is that we generally handle a set of alternative typings by separately typing conditional or case branches with the rest of the program. Another difference is that our type system introduces error types to allow expressions to be typed completely rather than aborting the typing process when a type error is encountered.

Whatever the ordering, unification-based approaches suffer in principle from a bias that results from the order in which substitutions are constructed and refined. Thus one can always find programs for which any particular approach performs rather poorly. Type error slicing tries to avert this problem by showing all the locations that contribute to type errors. A problem with this approach is that the wealth of provided information might become overwhelming. A number of different slicing techniques have been proposed [5, 9, 12, 22, 25], These all differ in their details, the features of type systems addressed (for example, [5] does not deal with let polymorphism), and the efficiency of the implemented methods. Otherwise, the produced slicing results are all very similar.

From a technical perspective, Kustanto and Kameyama's unification-based slicing approach [12] is most closely related to lazy typing. They use special unification variables ψ to record the conflicts between the types that ought to be unified. Thus their unification algorithms, like ours, doesn't terminate in presence of non-unifiable type equations. An important difference, however, is that when two types can't be unified, they are merged into a ψ variable in their approach, whereas in our approach a typing pattern is derived that represents the location of type errors and that can introduce appropriate $\perp$ types into the result type.

Lazy typing is also related to the work on discriminative sum types (also called soft typing) to locate type errors [19, 20]. The technical differences discussed in Section 1.2 lead to different typing behaviors for the two approaches. First, soft typing extracts all locations involved in type errors and is thus essentially a type-error slicing approach, whereas lazy typing blames the most likely error location when the context supports such a judgment. Second, lazy typing provides change suggestions in some cases, whereas soft typing, like all error-slicing approaches, does not. Finally, error locations reported by soft typing may contain program fragments that have nothing to do with type errors. For example, a variable used for passing type information only will be reported as a source of type errors if it is unified once with some sum types during the type inference process. In lazy typing, on the other hand, only locations that contribute to type errors are reported.

Since lazy typing finds most likely error locations and suggests expected types for erroneous locations, the question arises how our method compares to program repairing approaches [10, 11, 15, 17]. McAdam built an algorithm on top of the algorithm $\mathcal{W}$ to suggest program changes by designing a unification algorithm modulo linear isomorphism [17]. Heeren [10] has pointed out the potential problems with that approach, including that the suggested fix may lead to the transfer of type inconsistency to some other place. Lazy typing does not suffer from this problem; it guarantees that one type error will be eliminated if the error source is changed to an expression of the suggested type.

The Top framework [10] suggests program changes in several steps. First, the type constraints for expressions are collected and ordered. Second, the constraints are solved to decide the types for expressions. If there are type errors, heuristics are employed to find the error locations. Instead of changing type checkers or compilers, Seminal [15, 16] improves error reporting by searching for a well-typed program that is similar to the ill-typed program. Seminal mainly consists of two steps. First, it locates a type error through top-down removal. Specifically, if a program is ill typed, Seminal tries to remove each top-level expression to check whether the removal of that expression will eliminate the type error. If this succeeds for a top-level expression, then seminal recursively searches within that expression. Second, for the located expressions, it performs constructive changes at that location. Examples of such changes are the removal of an argument from a function call, swapping the ordering of arguments to function calls, and so on. After new expressions are constructed, each one is type checked. Seminal then suggests changes to the users by ordering all the programs that passed type checking using some heuristics. When a suggested change becomes too large, for example changing the whole program to a variable, this indicates that there are multiple type errors in the program, and Seminal will enter the so-called "triage mode", in which when a top-level expression is removed together with its sibling expressions to remove some type constraints. This will improve the likelihood that the program will be well-typed. Seminal tries to find as many errors as possible.

Top and Seminal use heuristics to order program-change suggestions. We argue that contextual information is a very good heuristic to finding errors since contexts reflect how erroneous functions or expressions will be used. Thus lazy typing works best and produces good results when the contextual information is avail-

able. This is particularly true in cases when there is more than one type error. Our algorithm is also more efficient for the interactive use because variational type inference, by design, exploits common code parts and can work very efficiently with choice types [2]. In contrast, Seminal has to type each generated program separately.

Similarly to lazy typing, Johnson and Walz's unification-based approach also uses contextual information to help find more accurate error locations [11]. However, the two approaches use context information differently. In their approach, when a type variable has to be unified with two conflicting types, the type variable is mapped to a disjunction of these two types, and the typing process continues. The conflict is eventually resolved by something like "usage voting", that is, whatever type a variable is unified with most often, will be selected. In contrast, lazy typing will map a type variable to whatever type will cause the whole program to be well-typed, because that is the ultimate goal of type checking and type-error recovering. The use of choice types allows us to explore the combinations of different possibilities efficiently.

Finally, various methods have been proposed to explain the type inference process. Chameleon can generate explanation about why there is a type error, why a function or expression has a specific type and so on [23, 24]. Chitil observed that the primary obstacle to understanding why a program has a type error or why an expression has a specific type is the lack of compositional explanations [4]. Since computing principal types is not compositional but principal typing is, Chitil proposed the concept of compositional explanation graphs, based on which programmers can freely navigate between different parts of the typing tree and gain a better understanding of the types for particular expressions.

7. Conclusions

We have presented *lazy typing*, a new approach to generating type error messages. Our method is based on delaying unification and error decisions of the type system through the use of choice and error types. Choice types represent delayed decisions in alternatives, and the nesting of choice types provides rich context information that can be effectively exploited for error localization. Choice types are analyzed at the end of the typing process. In many cases the structure of choice types also facilitates the derivation of type-change suggestions for expressions containing type errors.

We have shown the correctness of lazy typing with regard to the traditional Hindley-Milner system by demonstrating that the systems produce the same results in the case of type-correct programs, and that lazy typing will neither report type errors for correct programs, nor report type-incorrect programs as type correct. The practical evaluation and comparison with related tools has showed that lazy typing is very effective and can report errors in most cases with high precision. In most tested cases, lazy typing performed as well or better than other tools.

References

[1] F. Baader and W. Snyder. Unification Theory. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, chapter 8, pages 445–533. Elsevier Science Publishers, Amsterdam, NL, 2001.

[2] 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.

[3] S. Chen, M. Erwig, and E. Walkingshaw. Extending Type Inference to Variational Programs. Technical report, School of EECS, Oregon State University, 2012. submitted for publication.

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

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

[6] V. Choppella and C. T. Haynes. Diagnosis of ill-typed programs. Technical report, Indiana University, 1995.

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

[8] 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.

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

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

[11] 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.

[12] C. Kustanto and Y. Kameyama. Improving error messages in type system. *Information and Media Technologies*, 5(4):1241–1254, 2010.

[13] 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.

[14] 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.

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

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

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

[18] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[19] 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.

[20] M. Neubauer and P. Thiemann. Haskell type browser. In *ACM SIGPLAN Workshop on Haskell*, pages 92–93, 2004.

[21] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, Jan. 1965.

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

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

[24] P. J. Stuckey, M. Sulzmann, and J. Wazny. Improving type error diagnosis. In *ACM SIGPLAN Workshop on Haskell*, pages 80–91, 2004.

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

A. Type Inference

The type inference for the lazy typing system consists of two steps. The first step is a minor modification of algorithm $\mathcal{W}$ [7], which returns a variational type, a unifier that maps type variables to variational types, and an environment which contains all the delayed constraints. The second step consists of a post-unifying process, in which we try to resolve all variational types by unifying all the alternatives in each choice type.

The algorithm for the first step deviates from algorithm $\mathcal{W}$ in three aspects:

(1) Instead of using the traditional *robinson* [21] unification algorithm, it uses the partial variational unification algorithm developed in [2], which has been shown to compute most general and most defined types, which means that in the case of that two variational types don't unify, as few as possible type errors are introduced into the type.

(2) The conditional branches and case alternatives are types "symmetrically", that is, type constraints introduced in the earlier branch or alternative will not affect the typing for a later branch or alternative.

(3) Instead of unifying types among conditional branches and case alternatives eagerly, we delay these unification problems and collect them in an environment Δ, which will be resolved in the second step.

The algorithm is almost a literal translation from the typing rules in Figure 3. An excerpt of the algorithm is shown at the top in Figure 8. We use *pvunify* to denote the partial variational unification algorithm. We use $\theta_2\theta_1$ to denote the composition of two unifiers θ_2 and θ. The operation $\theta = \theta_2 \uplus_D \theta_3$ computes θ_1 by combining θ_2 and θ_3 as follows. If $\alpha \mapsto \phi$ appears only in θ_2 or θ_3, then it is copied into θ. If $\alpha \mapsto \phi_2$ occurs in θ_2 and $\alpha \mapsto \phi_3$ occurs in θ_3, then θ will have $\alpha \mapsto D\langle\phi_2, \phi_3\rangle$.

To infer the type for an `if` expression, we first infer the type for the condition and the two branches under the updated environment. Note that the types for two branches are inferred independently. To avoid the unifiers for these two branches to interfere with each other, we create a new unifier that encapsulates the conflicts of type mapping with choice types. We then unify the type for the condition with `Bool`, which returns a unifier and typing pattern, to indicate which alternatives are unifiable and which aren't. We then mask the type for the branches, represented by a choice type with two alternatives for the corresponding branches, with the typing pattern, which will replace all $\top$s with that particular type and leave all $\bot$s unchanged. The result of the masking is returned as the result type for the `if` expression. The delayed constraint environment and unifiers are composed appropriately and returned.

For example, consider inferring the type of the following expression with $\Gamma = \{\mathtt{x} \mapsto \alpha\}$.

$$e_1 = \texttt{if x then odd x else not x}$$

After typing the condition `x`, we have $\phi_1 = \alpha$, $\theta_1 = \varnothing$, and $\Delta = \varnothing$. After typing the first branch `odd x`, we get $\phi_2 = \texttt{Bool}$, $\theta_2 = \{\alpha \mapsto \texttt{Int}\}$, and $\Delta_2 = \varnothing$. After typing the second branch `not x`, we have $\phi_3 = \texttt{Bool}$, $\theta_3 = \{\alpha \mapsto \texttt{Bool}\}$, and $\Delta_3 = \varnothing$. Assuming a fresh D, we obtain $\theta = \theta_2 \uplus_D \theta_3 = \{\alpha \mapsto D\langle\texttt{Int}, \texttt{Bool}\rangle\}$. Variational unification of $\theta(\phi_1) = D\langle\texttt{Int}, \texttt{Bool}\rangle$ with `Bool` gives the typing pattern $D\langle\bot, \top\rangle$ and $\theta_4 = \varnothing$. Now the return type is $D\langle\bot, \top\rangle \lhd D\langle\texttt{Bool}, \texttt{Bool}\rangle = D\langle\bot, D\langle\texttt{Bool}, \texttt{Bool}\rangle\rangle$, which is $D\langle\bot, \texttt{Bool}\rangle$ by eliminating dead alternatives. Likewise, the result delayed constraint environment is $\{D\langle\bot, \texttt{Bool}\rangle\}$. Inspecting the result type shows that there is a type error in the left branch of the `if` statement.

In the second step, we compute a unifier that maps type variables to monotypes, while trying to eliminate choice types as much as possible. We are now essentially deciding which of the delayed type constraints can be solved. If all the constraints can be solved and the choice types can be simplified to monotypes, there is no type error in the expression. Otherwise, we will call the error reporting algorithm to generate type error messages. The algorithm for the second step, *infer2* is presented in the middle of Figure 8, where ζ denotes a unifier whose codomain are only monotypes.

$$\boxed{\mathit{infer1} : \Gamma \times e \to \phi \times \theta \times \Delta}$$

$\mathit{infer1}(\Gamma, \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3) =$
  $(\phi_1, \theta_1, \Delta_1) \leftarrow \mathit{infer1}(\Gamma, e_1)$
  $(\phi_2, \theta_2, \Delta_2) \leftarrow \mathit{infer1}(\theta_1(\Gamma), e_2)$
  $(\phi_3, \theta_3, \Delta_3) \leftarrow \mathit{infer1}(\theta_1(\Gamma), e_3)$
  D is fresh
  $\theta \leftarrow \theta_2 \uplus_D \theta_3$
  $(\pi, \theta_4) \leftarrow \mathit{pvunify}(\theta(\phi_1), \texttt{Bool})$
  return $(\theta_4(\pi \lhd D\langle\phi_2, \phi_3\rangle), \theta_4\theta\theta_1,$
    $\theta_4(\theta(\Delta_1) \cup \Delta_2 \cup \Delta_3 \cup (\pi \lhd D\langle\phi_2, \phi_3\rangle)))$)

$$\boxed{\mathit{infer2} : \phi \times \theta \times \Delta \to \phi \times \theta \times \Delta}$$

$\mathit{infer2}(\phi, \theta, \Delta) =$
  $\zeta \leftarrow \mathit{uni}(\phi \doteq \phi \cup \{\phi_1 \doteq \phi_1 | \phi_1 \in \Delta\} \cup \{\phi_2 \doteq \phi_2 | \alpha \mapsto \phi_2 \in \theta\})$
  return $(\zeta(\phi), \zeta(\theta), \zeta(\Delta))$

$$\boxed{\mathit{uni} : \mathcal{R} \to \zeta} \quad \mathcal{R} = \{\phi \doteq \phi\} \quad \zeta = \{\alpha \mapsto \tau\}$$

$$\begin{aligned}
\mathit{uni}(\tau_1 \doteq \tau_2 \cup \mathcal{R}) &= \begin{cases} \mathit{uni}(\zeta(\mathcal{R}))\zeta & \mathit{robinson}(\tau_1, \tau_2) = \zeta \\ \mathit{uni}(\mathcal{R}) & \mathit{robinson} \text{ fails} \end{cases} \\
\mathit{uni}(\bot \doteq \phi \cup \mathcal{R}) &= \mathit{uni}(\mathcal{R}) \\
\mathit{uni}(\phi \doteq \bot \cup \mathcal{R}) &= \mathit{uni}(\mathcal{R}) \\
\mathit{uni}(\phi_1 \doteq D\langle\overline{\phi}\rangle \cup \mathcal{R}) &= \mathit{uni}(\phi_1 \doteq \overline{\phi} \cup \mathcal{R}) \\
\mathit{uni}(D\langle\overline{\phi}\rangle \doteq \phi_1 \cup \mathcal{R}) &= \mathit{uni}(\overline{\phi} \doteq \phi_1 \cup \mathcal{R}) \\
\mathit{uni}(T\ \overline{\tau_1} \doteq T\ \overline{\tau_2} \cup \mathcal{R}) &= \mathit{uni}(\overline{\tau_1} \doteq \overline{\tau_2}) \\
\mathit{uni}((\phi_1 \to \phi_2 \doteq \phi_3 \to \phi_4) \cup \mathcal{R}) &= \mathit{uni}(\{\phi_1 \doteq \phi_3, \phi_2 \doteq \phi_4\} \cup \mathcal{R}) \\
\mathit{uni}(\alpha \doteq \phi) &= \begin{cases} \mathit{uni}(\zeta(\alpha) \doteq \tau \cup \zeta(\mathcal{R}))\zeta & \exists \tau = \downarrow \zeta(\phi) \\ \mathit{uni}(\zeta(\mathcal{R}))\zeta & \text{otherwise} \end{cases} \\
&\quad \text{where } \zeta = \mathit{uni}(\phi \doteq \phi) \\
\mathit{uni}(\phi_1 \doteq \phi_2 \cup \mathcal{R}) &= \mathit{uni}(\mathcal{R}) \\
\mathit{uni}(\varnothing) &= \varnothing
\end{aligned}$$

Figure 8: Type Inference and Post Unification Algorithm

The algorithm *uni* plays a fundamental role in the second step of type inference. For the unification problem $\phi_1 \doteq \phi_2$, *uni* returns a unifier ζ such that $\zeta(\phi_1) = \zeta(\phi_2)$ so that they can be reduced to monotypes. We use the notation $\phi_1 \doteq \phi_2 \cup \mathcal{R}$ to single out the unification problem $\phi_1 \doteq \phi_2$ and bind all others to $\mathcal{R}$.

Most of the computation rules are straightforward. When unifying two monotypes, we call the *robinson* algorithm to proceed. If it succeeds, we just use that result; otherwise we simply ignore the current unification problem. Unification problems with $\bot$on either side are ignored because we can't derive any constraint from them. When unifying a type against a choice type, then that type will be unified against each alternative of the choice type. Unifying type constructors T and $\to$ results in the unifying of all corresponding type pairs. There is a subtlety in unifying a type variable against a variational type ϕ. The unification succeeds if we can simplify ϕ to a monotype, which is realized by unifying ϕ with itself and then simplify the result. We write $\downarrow\phi$ for the reduction of a variaitional type to its normal form using the $\rightsquigarrow$ reduction relation. For example, given $\alpha \doteq D\langle\beta, \texttt{Int}\rangle$, we first solve the problem

$D\langle\beta, \texttt{Int}\rangle \doteq D\langle\beta, \texttt{Int}\rangle$ as follows.

$$\begin{aligned}
\zeta &= \mathit{uni}(\{D\langle\beta, \texttt{Int}\rangle \doteq D\langle\beta, \texttt{Int}\rangle\}) \\
&= \mathit{uni}(\{\beta \doteq D\langle\beta, \texttt{Int}\rangle, \beta \doteq D\langle\beta, \texttt{Int}\rangle\}) \\
&= \mathit{uni}(\{\beta \doteq \beta, \beta \doteq \texttt{Int}, \beta \doteq D\langle\beta, \texttt{Int}\rangle\}) \\
&= \mathit{uni}(\{\beta \doteq \texttt{Int}, \beta \doteq D\langle\beta, \texttt{Int}\rangle\}) \\
&= \mathit{uni}(\{\beta \mapsto \texttt{Int}\}(\{\beta \doteq D\langle\beta, \texttt{Int}\rangle\}))\{\beta \mapsto \texttt{Int}\} \\
&= \mathit{uni}(\{\texttt{Int} \doteq D\langle\texttt{Int}, \texttt{Int}\rangle\})\{\beta \mapsto \texttt{Int}\} \\
&= \mathit{uni}(\{\texttt{Int} \doteq \texttt{Int}, \texttt{Int} \doteq \texttt{Int}\})\{\beta \mapsto \texttt{Int}\} \\
&= \mathit{uni}(\{\texttt{Int} \doteq \texttt{Int}\})\{\beta \mapsto \texttt{Int}\} \\
&= \{\beta \mapsto \texttt{Int}\}
\end{aligned}$$

Now since $\downarrow(\{\beta \mapsto \texttt{Int}\}D\langle\beta, \texttt{Int}\rangle) = \texttt{Int}$, we have $\mathit{uni}(\alpha \doteq D\langle\beta, \texttt{Int}\rangle) = \mathit{uni}(\{\alpha \doteq \texttt{Int}\})\{\beta \mapsto \texttt{Int}\} = \{\alpha \mapsto \texttt{Int}, \beta \mapsto \texttt{Int}\}$.

With *uni* we can decide if a type ϕ can be turned into a monotype by simply testing if $\downarrow(\mathit{uni}(\phi \doteq \phi)(\phi))$ is a monotype. For example, if $\phi = D\langle\beta, \texttt{Int}\rangle$, then as we have seen before $\mathit{uni}(D\langle\beta\rangle\texttt{Int} \doteq D\langle\texttt{Int}, \alpha\rangle) = \{\beta \mapsto \texttt{Int}\}$. Applying the result to ϕ, it can be simplified to `Int`.

Now we illustrate the whole type inference process for the following expression.

$$e_2 = \texttt{(if True then f else g) x}$$

We use the following assumptions.

$$\begin{aligned}
&\texttt{f} : \texttt{Int} \to \beta \\
&\texttt{g} : \alpha \to \texttt{Bool} \\
&\texttt{x} : \alpha
\end{aligned}$$

Following the type inference algorithm for the `if` construct, we can do a similar analysis as we did for e_1 and obtain the following result.

$$\begin{aligned}
\phi &= D\langle\beta, \texttt{Bool}\rangle \\
\Delta &= \{D\langle\texttt{Int} \to \beta, \chi \to \texttt{Bool}\rangle\} \\
\theta &= \{\alpha \mapsto D\langle\texttt{Int}, \chi\rangle\}
\end{aligned}$$

Here χ is a fresh type variable.

In the second step, we first try to bring all the variational types to monotypes. Using the algorithm in Figure 8, we will solve the following unification algorithm.

$$\begin{aligned}
\mathit{uni}(\{\quad & D\langle\beta, \texttt{Bool}\rangle \doteq D\langle\beta, \texttt{Bool}\rangle, \\
& D\langle\texttt{Int} \to \beta, \chi \to \texttt{Bool}\rangle \doteq D\langle\texttt{Int} \to \beta, \chi \to \texttt{Bool}\rangle, \\
& D\langle\texttt{Int}, \chi\rangle \doteq D\langle\texttt{Int}, \chi\rangle \quad \})
\end{aligned}$$

We obtain the following monounifier $\zeta = \{\beta \mapsto \texttt{Bool}, \chi \mapsto \texttt{Int}\}$. Applying ζ to ϕ Δ, and θ, followed by simplification, we know the result type is `Bool`, Δ becomes empty, and the unifier becomes $\{\alpha \mapsto \texttt{Int}, \beta \mapsto \texttt{Bool}, \chi \mapsto \texttt{Int}\}$. Thus, we conclude there is no type error in e_2.

B. Programs Used in the Evaluation

The `fib` example.

```
f x = case x of
       0 -> [0]
       1 -> 1

plus :: Int -> Int -> Int
plus = (+)

fib x = case x of
      0 -> f x
      1 -> f x
      n -> fib (n-1) `plus` fib (n-2)
```

The `split` example.

```
split xs = case xs of
           [] -> ([],[])
           [x] -> ([], x)
           (x:y:zs) -> let (xs, ys) = split zs
                       in (x:xs, y:ys)
```

The `map` example.

```
map1 f [] = []
map1 f (x:xs) = f x : map1 f xs
test = map1 True "abc"
```

The `add3` example.

```
(\x->x+3) (if True then False else 1)
```

The `insert` example.

```
insert x [] = x
insert x (y:ys) | x > y = y : insert x ys
                | otherwise = x : y : ys
```

The `strlist` example.

```
add str lst
    | str `elem` lst = lst
    | True           = str:lst

v = add  ["error"] "location"
```

The `strlist1` example.

```
add str lst
    | str `elem` lst = [lst]
    | True           = str:lst

v = add  "error" ["location"]
```

The `if1` example.

```
h = if True then (\f-> f (f 2))
           else (\g-> g (g True))
```

The `if2` example.

```
v = let h = if True then (\f-> f (f 2))
                    else (\g-> g (g True))
    in h not
```

The `plus` example.

```
let x = 3 + true in 4+"hi"
```

The `fibbool` example.

```
f x = case x of
       0 -> [0]
       1 -> 1

plus :: Int -> Int -> Int
plus = (+)

fib x = case x of
      0 -> f x
      1 -> True
      n -> fib (n-1) `plus` fib (n-2)
```

The `condfun` example.

```
f x = if True then not x else x + 1
g x = if True then not x else 2
v   = f 3 + g True
```

The `fiblist` example.

```
f x = case x of
       0 -> [0]
       1 -> 1

fib x = case x of
      0 -> f x
      1 -> f x
      n -> head (f x)
```