On Repairing Ill-Typed Expressions

Tihomir Gvero[1], Ivan Kuraj[1], and Ruzica Piskac[2]

[1] École Polytechnique Fédérale de Lausanne (EPFL), Switzerland
[2] Yale, USA

Abstract. When developing code, a programmer typically knows the approximate structure of the desired expression. However, often the first attempt at writing it down results in an ill-typed code fragment. We propose an approach that automatically repairs code expressions based on the provided almost-correct code. Such a code repair can be applied in interactive scenarios like advanced code completion, as well as in automated repair in the compilation process.
We formally define the problem of automatically repairing ill-typed expressions. For the certain class of problems we describe a polynomial time synthesis algorithm that returns the best well-typed expression corresponding to the given ill-typed expression. We also present a complete algorithm that takes as input an ill-typed expression and returns the desired number of type-correct expressions that are closest to the input expression. We simultaneously fix all the type errors in the expression.

Keywords: software repair, corrections, type-checking, synthesis

1 Introduction

Software development provides a high degree of freedom and many different approaches can be adopted for writing code, yet when writing a program, the developer needs to follow the strict rules determined by the programming language. While coding, the developer often knows the approximate structure of the desired expressions but still may write code that does not compile, because some fragments are ill-typed. Such mistakes occur mainly because the developer does not know, by heart, how to choose and properly combine all the necessary declarations visible from the scope[3]. Moreover, modern libraries often evolve into complex application programming interfaces (APIs) that provide a large number of declarations. For this reason it is hard, if not impossible, to learn the specifics of the declarations and their utilization.

In a typical scenario when code does not compile, the compiler outputs an error message with the expression that is at the source of the error. Still, on many occasions the written expression reflects the intended structure of the code. In this paper we define an algorithm that automatically repairs code expressions based on the hinted structure of the ill-typed code. It finds well typed expressions

[3] We use the term "declarations" to refer to all the elements visible in the scope, such as variables, functions, and class hierarchy declarations

that are as close as possible to the given (potentially) ill-typed expression - we call such an input expression, a *backbone* expression. The algorithm can be applied in interactive scenarios like IDE code completion, to rank expressions based on their similarity to ill-typed code. Preferably, the best suggestions will fix the code keeping its overall structure. Another application is in providing automated repair in the compilation process.

In our previous work we developed a tool called InSynth [3] that automatically synthesizes code snippets based on the given type constraints. InSynth considers all user-defined declarations, together with any imported API calls, when performing the synthesis. In principle, a naïve algorithm for code repair based on InSynth could to solve the repair problem. The algorithm would first extend the initial environment with all type declarations that could be derived from the given ill-typed expression. Using the new environment we would run the InSynth algorithm. Because the InSynth algorithm is complete, eventually it will generate expressions (if they exist) following the structure of the given backbone expression. However, this may not be practical because the type inhabitation problem is a PSPACE-complete problem and the ranks of resulting expressions is determined using heuristic that ignores given ill-typed expression. The aim of our approach is to design a better and more efficient algorithm that is well-suited for repairing ill-typed expressions.

The input to our algorithm is an ill-typed, backbone expression. We propose two different algorithms. The first algorithm generates one well-typed expression that strictly follows the structure of the input expression. It decomposes the problem into finding connections between individual symbols in the backbone expression. The connections are build using a given set of repair declarations. Each connection represents a partial expression. We find the smallest such expressions using a weights mechanism. When we find all the partial expressions, we combine them following the original structure of the backbone expression. The algorithm simultaneously fixes all type errors in a given expression. The second algorithm searches for the best solutions, not only following the initial backbone expression, but also creating new ones that are mutations of the original one. This might lead us to a set of more interesting solutions in the case when the developer does not provide us with the best initial backbone expression. Additionally, this approach allows completeness in sense that we are able to generate all expressions with the given type, similarly as in InSynth, but now ranked by similarity to the given backbone expression. The algorithm also implements A* search that steer us towards the most desirable solutions.

A research on improving the software development process covers a large number of topics such as an automated program repair [7, 21, 14], enhancements of compilation process messages [1, 4, 9], and providing assistance to developers through inference of code [3, 11, 5, 6, 15]. As a result, a vast number of tools was created around a common high-level goal of facilitating software development. The manner in which such tools operate can be roughly divided into two categories: (1) as automated processes within the compiler (2) or as development assistants that require a certain level of interaction, usually through an IDE

interface. Many of the techniques behind these tools such as parsing error recovery by altering the input [1], a heuristic search for syntactically correct terms [15], a modification of abstract syntax trees and types [9], and an inference of semantically correct code fragments [5], share common insights. Motivated by the advances in both the theory of programming languages and techniques that are foundations of tools for software development, our approach addresses the problem of code repair from a new perspective, by providing an algorithm that extends existing and incorporates new ideas. The reason why our approach goes beyond the existing line of work is three-fold: (1) the approach tries to solve more general code repair problems constrained with the structure of given ill-typed terms, (2) it focuses on repairing programs in as much accurate way as possible according to the given hint and weight heuristics, while providing useful theoretical guarantees about the utilized repair algorithms, (3) it is fitting for realization as both an interactive and automated software development tool.

The contributions of this paper are:

- We formulate the problem of repairing ill-typed expressions. As input to the problem we take a backbone expression and the set of repair declarations. We introduce weak long normal form that allows a systematic search and a construction of the well-typed expressions. We identify special symbols used to extend the expressiveness of the input.
- We propose a novel repair calculus that specifies the rules that we use to derive a well-typed term from a backbone expression. The calculus describes how to fix a term using declarations with multiple arguments.
- We propose an algorithm that finds the best well-typed expression based on weights system introduced in [3]. We show that the algorithm is polynomial given a certain class of ill-typed expressions.
- We present a sound and complete algorithm that takes the ill-typed expression and finds a set of best solutions. The algorithm is based on A* search.

2 Motivating Examples

We begin with a series of examples that show how to apply our algorithm in practical software development. These examples are written in the Java programming language. As formally defined in Sec. 3, as input our algorithm takes terms in applicative long normal form, which closely corresponds to expressivity of the Java programming language. To emphasize the significance in practical software development, we choose examples from a set of the real code examples (most of them featured at `http://www.java2s.com/`).

2.1 Sequence of streams

We start by showing how our algorithm handles a given backbone expression that does not compile. This example is similar to the example presented in the InSynth paper [3], so this way we can compare and demonstrate the differences between InSynth and our algorithm. Consider the following program fragment:

```
import java.io.*;

public class Main {
  public static void main(String args[]) throws IOException {
    String body = "email.txt";
    String sig = "sign.txt";

    SequenceInputStream seqStream = new SequenceInputStream(body, sig) // error
    // ..., rest of the code is omitted
  }
}
```

The developer declared the variable seqStream; however, the specified expression assigned to seqStream does not compile. Still, from this backbone expression we can recognize the structure of the intended expression: our algorithm should construct an expression that preserves the relative position of the declarations from SequenceInputStream(body, sig). In the resulting expression a SequenceInputStream constructor should be used, with arguments that contain body and sig variables in their corresponding sub-expression trees. Our repair algorithm finds all such expressions, constructed from the declarations visible in the scope of the backbone expression. The found expressions are well-typed and ranked according to a metric that characterizes the resemblance to the starting backbone expression. The returned expression with the highest rank is

```
SequenceInputStream seqStream = new SequenceInputStream(
  new FileInputStream(body), new FileInputStream(sig))
```

This expression represents exactly the desired expression. When we ran [3] on the exactly same example, but without the backbone expression, the desired expression was ranked as the second highest. Our new algorithm outperformed InSynth on this example, showing that the backbone expression can increase the quality of returned results by InSynth.

An additional advantage that our repair algorithm has over InSynth is handling of constants. As an illustration, given the backbone expression:

```
SequenceInputStream seqStream = new SequenceInputStream("email.txt", "sign.txt")
```

our algorithm returns, as expected:

```
SequenceInputStream seqStream = new SequenceInputStream(
  new FileInputStream("email.txt"), new FileInputStream("sign.txt"))
```

In InSynth we were not able to synthesize code snippets with arbitrary literals. This way our code repair algorithm can be considered as an improved synthesis algorithm, because it also incorporates explicitly given literals in the final snippets.

2.2 Use of coercion functions

Many programming languages support coercion functions [18]. They are used for type conversion and can be applied automatically if needed, without the direct intervention from the developer. An automated insertion of coercion functions is

utilized for the purpose of fixing ill-typed expressions in many modern compilers, but usually in a limited manner (at most one coercion function can be used to fix an ill-typed expression). Our repair algorithm goes beyond this standard and allows more expressive transformations of ill-typed expressions. The algorithm is based on advanced methods for searching and adapting appropriate functions, combined with synthesizing any additional necessary arguments.

Consider the following code, in which the developer declares a byte buffer and wants to construct an expression of the type InputStream by merely hinting the desired type and usage of the declared buffer b:

```
import java.io.*;

public class Main {
  public static void main(String args[]) throws IOException {
    int off = 8, len = 512, size = 1024;
    byte b[] = args[0].getBytes();

    InputStream input = b; // error
  }
}
```

To repair the expression that initialize the variable input, we insert a coercion function (here represented as a constructor application):

```
InputStream input = new ByteArrayInputStream(b);
```

Our algorithm returns this expression as the highest ranked expression: the expression is well-typed, follows the simple structure of the backbone expression b, with the smallest size. The upcasting done in this case can be seen as an implicit insertion of a coercion function, which casts ByteArrayInputStream to its superclass, InputStream.

Our repair algorithm finds additional well-typed expressions, such as

```
InputStream input = new ByteArrayInputStream(b, off, len);
```

This expression also correctly repairs the given backbone expression, but it no longer represents a simple coercion function insertion. It is the ByteArrayInputStream overloaded constructor with three arguments. To create this expression, our algorithm considers a broader range of available functions and finds appropriate expressions that fill the places of the missing arguments. Those arguments are synthesized whenever a type-conversion function requires additional parameters.

2.3 Mutations of ill-typed expressions

Sometimes the developer writes an ill-typed expression that less reflects the structure of the desired expression, usually by giving a wrong order or number of arguments. Consider the following code that extensively uses calls to the standard Java API library to manipulate streams. The given backbone expression expresses a users wish to read a file compressed with the ZLIB library through a buffered stream. To read the file a user would need an InputStream object.

```
import java.io.*;
import java.util.zip.*;

public class Main {
  public static void main(String args[]) throws IOException {
    int buffSize = 1024, compLevel = Deflater.BEST_SPEED;
    String fileName = "compressed.txt";

    InputStream input = new BufferedInputStream(buffSize,
      new DeflaterInputStream(new FileInputStream, compLevel, true)); // error
  }
}
```

However, a user writes arguments of the BufferedInputStream constructor in a wrong order. In such a scenario our algorithm enumerates all the mutations of the abstract syntax tree given by the backbone expression. The algorithm then searches exhaustively all possible valid expressions considering a large number of different syntax trees. Those abstract syntax trees effectively correspond to useful modifications at the source code level, such as reordering, deletion or insertion of function arguments.

To create a desired expression our algorithm has to change the order of the BufferedInputStream arguments in the initial backbone expression. After applying a set of further modifications to the arguments we arrive to the new backbone expression. In the last phase we fix the backbone expression inserting Deflater constructor. The correct expression looks like:

```
InputStream input = new BufferedInputStream(new DeflaterInputStream(
  new FileInputStream(fileName), new Deflater(compLevel, true)), buffSize);
```

Each mutation of an initial backbone expression introduces additional penalty in terms of the resulting rank.

3 Preliminaries

Let B be a set of basic types. Types are formed according to the following syntax:

$$\tau ::= \tau \to \tau \mid v, \quad \text{where } v \in B$$

Given a set of types T, a *type environment* Γ is a finite set $\{x_1 : \tau_1, \ldots, x_n : \tau_n\}$ of pairs of the form $x_i : \tau_i$, where x_i is a variable of a type $\tau_i \in T$. We call the pair $x_i : \tau_i$ a *type declaration*.

In [3] we showed that judgements in so called applicative long normal form play an important role in solving the type inhabitation problem. We recall the definition, and, in addition, we introduce the notion of weak applicative long normal form.

Definition 1 ((Weak) Applicative Long Normal Form). *A judgement $\Gamma \vdash e : \tau$ is in (weak) applicative long normal form if the following holds:*

- $e \equiv f e_1 ... e_n$*, where* $n \geq 0$
- $(f : \tau_1 \to \ldots \to \tau_n \to \tau) \in \Gamma$*, where* $\tau, \tau_1, \ldots, \tau_n \in B$
- $\Gamma \vdash e_i : \rho_i$ *where* $i = 1..n$ *are in (weak) applicative long normal form.*

The main difference between applicative long normal form and weak applicative long normal form is that in the weak version, types ρ_i and τ_i do not need to conform, i.e. $\rho_i = \tau_i$ does not need to hold. However, in both versions f has exactly the same number of arguments as indicated by its type declaration.

Figure 1 shows the type derivation rule used to derive terms in applicative long normal form. This calculus is slightly more restrictive than the standard

$$\textsc{App}\ \frac{f : \tau_1 \to \ldots \to \tau_n \to \tau \in \Gamma \quad \Gamma \vdash e_i : \tau_i \quad \tau, \tau_i \in B \quad i = 1..n}{\Gamma \vdash f(e_1 \ldots e_n) : \tau}$$

Fig. 1. A rules for deriving terms in applicative long normal form

applicative calculus: the App rule requires that only those functions present in the original environment Γ can be applied to terms.

Example 1. For the above calculus and the type environment $\Gamma = \{f : \tau_1 \to \tau_2 \to \tau_3, g_1 : \sigma_1 \to \tau_1, g_2 : \sigma_2 \to \tau_2, a : \sigma_1, b : \sigma_2\}$, $\Gamma \vdash f(g_1(a), g_2(b)) : \tau_3$ is in applicative long normal form, while $\Gamma \vdash f(a, b) : \tau_3$ is in weak applicative long normal form.

Definition 2 (A Backbone Expression). *Given a type environment Γ and a judgment $\Gamma \vdash e : \tau$ is in weak applicative long normal form, expression e is called a* backbone expression *in Γ.*

The main problem that we are addressing is finding an expression e for the given backbone expression e_b, such that $\Gamma \vdash e : \tau$ is in applicative long normal form, i.e. it is well-typed, and e follows the structure of e_b as close as possible. To solve the latter requirement, the structure of the abstract syntax tree e_b needs to correspond to the structure of e. If that holds, we say that e_b is a *minor* of e. Intuitively, it means that each edge in the abstract syntax tree of e_b represents a path in the abstract syntax tree of e. We denote the set of candidates for repair e_b with $\mathit{repair}(e_b, \Gamma)$:

$$\begin{aligned} \mathit{repair}(e_b, \Gamma) = \{e \mid & AST(e_b) \text{ is a minor of } AST(e), \\ & \Gamma \vdash e : \tau \text{ is in long applicative normal form}\} \end{aligned}$$

Since the set $\mathit{repair}(e_b, \Gamma)$ can contain a large number of elements, in order to rank them, we introduce a metric that measures the similarity between element in $\mathit{repair}(e_b, \Gamma)$ and their minor e_b.

Let $e_b = f(a, b)$ be a backbone expression and $\Gamma_b = \{f : \tau_1 \to \tau_2 \to \tau, a : \tau_3, b : \tau_4\}$ a set of declarations that appear in e_b. In the rest of the paper we will add prefix "b-" in front of terms: expression, environment and declaration to specify that they are determined by e_b. Therefore, e_b is a backbone expression or *b-exppression* for short, Γ_b is a b-environment and $f : \tau_1 \to \tau_2 \to \tau$ is a b-declaration. Similarly, we add prefix "r-" in front of an environment if it is $\Gamma_r = \Gamma \setminus \Gamma_b$, a declaration if it is in Γ_r and an expression if it contains only symbols from Γ_r. A b-expression can contain special symbols ?, $?_{arg}$ and $?_{head}$ which we use to mark holes in expressions. We do not put holes in Γ_b.

4 Applicative Repair Calculus

We modify the calculus in Figure 1 to favor expression that closely follow a structure of a given backbone expression. The idea is to expend the calculus such that information on backbone expression is propagated via rules using term $e_b : \tau \rightsquigarrow e$. The term argues that e is an expression that embeds the e_b backbone expression. The new calculus is presented in Figure 2.

$$\textsc{AppBone}\ \frac{f : \tau_1 \to \ldots \to \tau_n \to \tau \in \Gamma_b \quad \Gamma \vdash_r a_i : \tau_i \rightsquigarrow e_i \qquad \tau, \tau_i \in B \quad i = 1..n}{\Gamma \vdash_r f(a_1, \ldots, a_n) : \tau \rightsquigarrow f(e_1, \ldots, e_n)}$$

$$\textsc{AppSubHead}\ \frac{f : \tau_1 \to \ldots \to \tau_n \to \tau \in \Gamma_b \quad \Gamma \vdash_r a_i : \tau_i \rightsquigarrow e_i \qquad \tau, \tau_i \in B \quad i = 1..n}{\Gamma \vdash_r ?_{head}(a_1, \ldots, a_n) : \tau \rightsquigarrow f(e_1, \ldots, e_n)}$$

$$\textsc{AppSub}\ \frac{f : \tau_1 \to \ldots \to \tau_n \to \tau \in \Gamma \quad \Gamma \vdash_r \tau_? : \tau_i \rightsquigarrow e_i \qquad \tau, \tau_i \in B \quad i = 1..n}{\Gamma \vdash_r ? : \tau \rightsquigarrow f(e_1, \ldots, e_n)}$$

$$\textsc{AppPath}\ \frac{\begin{array}{c} f : \tau_1 \to \ldots \to \tau_n \to \tau \in \Gamma_r \\ \text{one argument:} \quad \Gamma \vdash_r g(a_1, \ldots, a_m) : \tau_p \rightsquigarrow e_p \\ \text{other } (n{-}1) \text{ arguments:} \quad \Gamma \vdash_r ?_{arg} : \tau_j \rightsquigarrow e_j \\ p, j \in \{1..n\} \quad \tau, \tau_i \in B \quad i = 1..n \end{array}}{\Gamma \vdash_r g(a_1, \ldots, a_m) : \tau \rightsquigarrow f(e_1, \ldots, e_n)}$$

Fig. 2. Rules for repairing terms in applicative long normal form

The calculus, denoted with relation $\vdash_r$ contains four rules:

- AppBone: This rule embeds b-declarations into final expression e. Can be applied if τ matches return type of symbol f.
- AppPath: This rule inserts r-declarations around b-declarations. Can be applied if τ matches return type of a r-declaration from Γ.

- AppSubHead: This rule repairs (substitutes) $?_{head}$ symbol with r-declaration. Can be applied if there is a r-declaration that has the same number of arguments as $?_{head}$ declaration.
- AppSub: This rule repairs (substitutes) ? symbol with any declaration from Γ. Can be applied if there is a declaration from Γ with the return type τ.

AppBone and AppPath are the most important rules. AppBone embeds b-declarations and AppPath builds paths between them. The path is constructed using r-declaration to connect two b-declaration. Only one argument e_p is used for path, the remaining (n-1) must be filled in with expressions constructed only using r-declarations. We use a special symbol $?_{arg}$ to mark them. They are constructed only by AppPath rule. Later in this section we define the path and those remaining arguments around the path.

So far beside regular declarations from Γ_b we allow $?_{arg}$ to appear in b-expression. This symbol denotes missing arguments. However, a user sometimes wants to omit declarations that appear at place of head symbol. When we say a head symbol, in this context, it is any head symbol of any subexpression in a b-expression. To allow this we introduce the $?_{head}$ symbol that specify missing (deeply nested) head symbol. For instance, $?_{head}(a, b)$ is a b-expression that misses a declaration with two arguments. The first argument must contain a symbol and the second b. Our goal is to find such declaration in Γ_r. This is the aim of the rule AppSubHead.

Finally, we introduce symbol ? that represents a missing argument that can be constructed using declarations from entire Γ. We use it to build a complete algorithm in Section 5.

Path Construction: The key point in repairing a backbone expression e_b is building partial expressions that connects the pieces of e_b. One partial expression is identified by one type-declaration pairs in e_b. We say that a declaration $x : \tau' \in \Gamma_b$ and a type τ form a pair if the symbol x appears at the place of an argument with the type τ in a b-expression. For instance, let $\Gamma_b = \{f_1 : \tau_1 \to \tau_2 \to \tau_3, f_2 : \tau_4, f_3 : \tau_5\}$, $e_b = f_1(f_2, f_3)$ be a b-expression and τ a desired type. Then, e_b contains three type-declaration pairs: $(\tau, f_1 : \tau_1 \to \tau_2 \to \tau_3)$, $(\tau_1, f_2 : \tau_4)$, $(\tau_2, f_3 : \tau_5)$.

To find if the partial expression exists we run the search in a backward manner starting from a type. For instance, we start from the desired type τ and try to reach f_1. In other words, we start from τ and construct a path by connecting r-declarations to reach f_1. (unless $\tau = \tau_3$, then f_1 is the only declaration on the path). Once we reach f_1 we discover two new pairs and try to build two new paths between τ_1 to f_2 and τ_2 to f_3. Finding such paths allows us to connect f_1 with f_2 and f_3. We use rule AppPath to perform this task.

However, r-declarations usually have more than one argument (f in AppPath has n arguments). Only one argument is used for building the path, where others are not directly on the path. We call them "boarding" arguments. Thus we need to find and build those arguments as well. Again, we use AppPath rule to build them. They are denoted with $?_{arg}$, and as mentioned they are constructed only using AppPath. This way we construct a partial expression that contains a path between a type and a b-declaration. The general form of such a path is given

by:

$$p_1({}^1 a_{\{1,1\}}, \ldots, p_2({}^2 \ldots, p_n({}^n a_{\{n,1\}}, \ldots, f({}^{n+1} b_1, \ldots, b_{m_{n+1}}), \ldots, a_{\{n,m_n\}}), \ldots), \ldots, a_{\{1,m_1\}})$$

where p_i are r-declarations and m_i is a number of arguments of p_i, $i = 1..n$. The index over the parentheses represents the depth at which it appears in the expression. The path from τ to f is a sequence of declarations $p_1, p_2, \ldots, p_n, f$. The expressions $a_{\{j,i\}}$ are "boarding" arguments and they contain only r-declarations, as mentioned before. The arguments b_i may contain some other paths.

4.1 Mutating B-expression

It might happen that a user writes an initial b-expression e_b that does not derive any expressions or only few interesting solutions. However, maybe a slight modification of e_b will leads us to interesting repairs. Thus, to be more flexible we allow b-expression to be modified. This produce new ones that guide the algorithm in other interesting directions.

Definition 3 (Strict repair). *An expression e with the type τ strictly follows a given b-expression e_b if $\Gamma_r, \Gamma_b \vdash_r e_b : \tau \rightsquigarrow e$.*

Definition 4. *An expression e with the type τ loosely follows a given b-expression e_b if $\Gamma_r, \Gamma_b \vdash_r e'_b : \tau \rightsquigarrow e$, where e'_b is constructed from the symbols that appear in e_b.*

However, if we set e'_b to e_b we see that two definitions become identical, and thus we can talk only about the level of looseness.

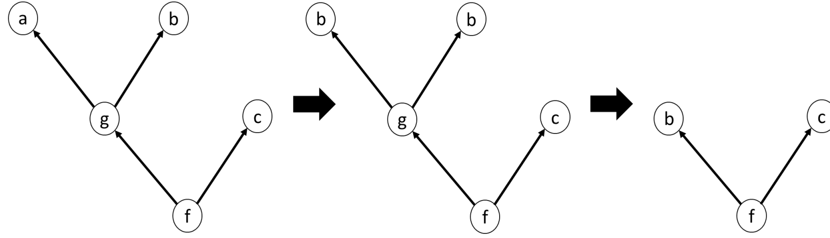

Fig. 3. Mutating a b-expression. Each transition represents one mutation.

To estimate this level we measure how far e'_b from e_b is. A number of transformations (mutations) we need to perform to go from e_b to e'_b determines the level. Higher the number, higher the level of looseness. A single mutation, substitutes a b-symbol in e_b with another symbol in $\Gamma_b \cup \{?\}$.

For instance, if original b-expression is $f(g(a, b), c)$, and if we want to mutate a one mutation substitutes it by b and we get $f(g(b, b), c)$. The next mutation, might substitute the second b to c, and the last one g with c (Figure 3).

4.2 Calculating Weights of a Partial Expression

We expend partial expression definition from [3]. We say that an expression is partial if it contains hole. In some sense it is incomplete expression. However, unlike in [3] we say that a hole can contain a b-expression. We use this extra information to substitute the hole with declarations that build an expression close to the containing b-expression.

In the previous work [3] we introduce the notion of weights. Initially, we assign weights to each symbol based on two metrics: the proximity of symbol definition to the point where programmer invokes the tool, and the frequency with which the symbol appears in the training data corpus. Smaller the weight, higher the priority of a declaration. We use the weights to guide the synthesis algorithm and to rank the final expressions.

In this paper, we use weights for the same purpose. However, we calculate a weight of a partial and a final expression in a new way. The difference is that we calculate a weight of a hole and add it to the total weight of the partial expression. The weight function w assigns a weight to a partial expression e in the following way:

- If $e = f(e_1 \dots e_n)$ and if $f \in \Gamma_r$ $w(e) = w(f) + \sum_{i=1}^{n} w(e_i)$
- If e is a hole $[f(e_1 \dots e_n)] : \tau$ then $w(e) = MinPathCost(\tau, f)$, where $MinPathCost$ return a weight of the smallest partial expression that contains τ and f.
- If e is an empty hole $[\,] : \tau$ then $w(e) = 0$.

Additionally, if there are mutations that are associated with e we add the cost of each mutation to the weight of the total expression e. A cost of a mutation is equal to the sum of the depth and the distance costs:

- **The depth cost** depends on the depth in the expression at which we perform mutation. Closer a mutation to the leaves smaller the cost. The intuition is that if a user made a mistake while inserting the original b-expression, then the mistake is small and it occurs near the leaves of the expression. In Figure 3 the cost is smaller if we mutate a then f.
- **The distance cost** depends on the distance in the initial b-expression between the new symbol and the old one. Closer the new symbol to the old one smaller the cost.

5 Algorithm

We illustrate and analyze the complexity of the two repair algorithms. The first algorithm generates an expression with the smallest weight that strictly follows a given e_b. The second, algorithm generates N expressions that loosely follows a given e_b.

5.1 Strict Repair with One Minimal Expression

We describe the algorithm that for a given environment Γ, a broken expression e_b and a desired type τ, finds an expression with the minimal weight that strictly follows the structure of e_b, if such an expression exists (Figure 4).

StirctRepair(e_b, τ, Γ):
1. Using e_b find Γ_b, and then $\Gamma_r = \Gamma \setminus \Gamma_b$
2. Instantiate e_b with declarations from Γ_r.
3. For each instantiation e'_b do:
   3.1. Find all pairs type−declaration in e'_b.
   3.2. For each pair $(\tau, b : \tau')$ do:
      3.2.1. Build a minimal partial expression that contains a path from τ to $b : \tau'$, using the function $MinPath(\tau, b : \tau', \Gamma_r)$.
   3.3. If for each pair $(\tau, b : \tau')$ such an expression exists, connect them into one expression strictly following the structure of e'_b, and fill in the $?_{arg}$ arguments with expressions from $Table_r$.
4. For each e'_b calculate the weight of its corresponding expression.
5. Find the minimal among them and return it as a result.

Fig. 4. The algorithm that finds the an expression that strictly generates given b-expression.

In general, e_b can contain a variable $?_{head}$. To make $?_{head}$ free e_b, we instantiate each $?_{head}$ with candidate r-declarations, in the step 2. A r-declaration is a candidate if the number of arguments that follows $?_{head}$, in a b-expression, matches the number of arguments of the r-declaration. For example, if b-expression is $?_{head}(a, b(?_{head}(c)))$ and in Γ_r we have $r_1 : \tau_1 \to \tau_2 \to \tau_3$ and $r_2 : \tau_4 \to \tau_5$. The first $?_{head}$ has two arguments, thus r_1 is a good candidate to substitute it. Similarly, the second can be substituted by r_2. Thus, instantiated b-expression is $r_1(a, b(r_2(c)))$. We treat it as any regular b-expression. Potentially, there can be many declarations that can satisfy $?_{head}$ symbol and thus many different instantiations of a single b-expression. Therefore, we also annotate Γ_r declarations with r-declaration symbols that are candidates for all $?_{head}$ in b-expression.

Next, we find type-declaration pairs in the step 3.1. The key step in the algorithm above is building minimal partial expression using MinPath function in Figure 5. We try to build a partial expression that contains a path from a type τ and reaches a b-declaration $b : \tau'$.

In the algorithm, we use the auxiliary function $retType$ that maps a type to its return type:

$$retType(\tau_1 \to \ldots \to \tau_n \to \tau) = \tau$$

When we obtain partial expressions we need to combine them to form an expression that embeds e_b, i.e., fallows its structure strictly. For instance, if we obtain $r_1(b_1([\,] : \tau_1), r_2) : \tau_0$ and $r_3(b_2) : \tau_1$ partial expressions for the pairs

above, then the expression that embodies $b_1(b_2)$ is $r_1(b_1(r_3(b_2)), r_2) : \tau_0$. It is obtained by substituting the hole $[\,] : \tau_1$ with $r_3(b_2)$.

MinPath(τ, $b : \tau'$, Γ_r):
1. We build an empty priority queue PQ that stores partial expressions.
   Smaller the weight of the partial expression, higher the priority in PQ.
2. We store hole $[\,] : \tau$ in PQ.
3. If PQ is empty we terminate.
4. We remove first partial expression exp from PQ.
5. Put its type in the set Visited.
6. If for the hole $h : \tau_h$ in exp holds that $\tau_h = retType(\tau')$,
   insert symbol b at the place of the hole, followed by new holes corresponding
   to each argument type of τ'. The holes are ordered exactly as they appear
   in τ'. Terminate the algorithm and return this new partial expression as a result.
7. If not, search for all declarations in Γ_r that return the type τ_h.
8. For each declaration do:
   8.1. If it has m arguments, create m different partial expressions, such
        that, each has only one hole (only one), but each has a hole at the different
        place (argument). The hole is directly on the potential path between τ and
        $b : \tau'$. We instantiate other arguments with types τ_i, with expressions
        $Table_r(\tau_i)$. Those are "boarding" arguments on the potential path.
   8.2. If the type of the hole is in Visited set we discard the partial expression,
        leaving m' expressions.
   8.3. We replace the hole $h : \tau_h$ in exp by remaining m' partial expressions,
        thus creating from exp new m' partial expressions.
   8.4. We calculate weight of each partial expression and insert it in PQ if its hole
        has a different type than any hole of any expression that is already in PQ.
   8.5. If there is an expression with such a hole already in PQ we compare its weight
        with the weight of the new partial expression. We leave in PQ one with
        the smaller weight.
9. We go to the step 3.

Fig. 5. The algorithm that finds a path from a type to a b-declaration.

In MinPath, if a type does not reach the $b : \tau'$ declaration then in the next iteration we try to reach it with the new type. We use one r-declaration per iteration to reach the new type. Maximal number of steps is therefore equal to the maximal number of different types in Γ_r. The path between τ and $b : \tau'$ is the bases the partial expression. To build the partial expression we also need to fill in "boarding". We pre-compute the minimal expressions using only declarations from Γ_r. We keep only one such expression per a different type in $Table_r$. We present the algorithm that builds this table in Figure 6.

In GeneratingTable, we first transform Γ_r declarations into individual partial expressions by filling their arguments with holes. In each iteration find the minimal one with the type τ, with no holes and for which $Table_r(\tau)$ is not occupied. We store it in the $Table_r$ and use it to substitute holes with type τ in the remaining partial expressions. This is the minimal representative of type τ

GenerteTable$_r(\Gamma_r)$:
1. For each declarations in Γ_r we fill in the arguments with holes that have the corresponding type argument.
2. We put them in the set Exp_r.
3. We remove complete expressions (with no holes) from Exp_r and put them into a priority queue PQ if their return type τ' has empty entry in $Table_r(\tau')$. If there are many complete expressions with the same return type, we put into PQ one with the smallest weight. PQ gives a higher priority to expressions with smaller weight.
4. If PQ is empty, we terminate.
5. We remove the first expression $exp : \tau$ from PQ.
6. Put exp into the entry $Table_r(\tau)$.
7. We substitute all holes with the type τ that appear in the expressions in Env_r.
8. We go to the step 3.

Fig. 6. The algorithm that generates minimal d-expressions used as "*boarding*$''$.

which means that it contributes the least to the new expressions with hole types τ. This way we continue creating smallest expressions.

Property: We prove that StrictReapair generates the smallest expression if one exists for a given b-expression, a desired type and Γ. Let us assume b-expression is instantiated. Key part is to prove that each partial expression that contains a path from τ and $b : \tau'$ is the smallest one:

1. We always choose the smallest "boarding" argument for a given argument type.
2. When we remove a partial expression exp_1 from PQ, with a hole type τ_1, it has the smallest weight among all partial expressions with a hole with the type τ_1.
   (a) When we remove exp_1 from PQ, it is the expression with the smallest weight in PQ.
   (b) An expression exp_2 synthesized after we remove exp_1 will contain a partial expression exp_3 from PQ (immediately after removing exp_1) or exp_1. Because the weight of exp_3 is greater of equal to exp_1, exp_2 must have strictly greater weight than exp_1. That is why we keep set Visited to discard such expressions on time.
3. Building partial expressions in the step 8 using Γ_r we reach all hole types reachable from the τ, before we reach τ'.

From the above we conclude that we indeed generate the smallest partial expressions for a pair. If those expressions exist for each pair, then the weight of the expression is equal to the sum of the weights of those expressions plus the weights of missing "boarding" arguments that substitutes all $?_{arg}$ symbols. Therefore, the weight of the final expression will be the smallest one. Finally, we choose the smallest one among all expressions that corresponds to different e_b instantiations. Similarly, we prove that $Table_r$ contains only expressions with the smallest weights for a given type.

Complexity: In the step 8 of the MinPath algorithm, for a type τ_i we select declarations from Γ_r, that return type τ_h. Let us say there are n_i such declarations. Note that those declarations return type τ_i. Let us assume we have selected k types. Each type is selected at most once due to Visited set. This means that $(\sum_{i=1}^{k} n_i) \leq n$, where n is the number of all declarations. Let's m be the maximal number of arguments for all declarations. This means there will be at most $m * n$ new partial expressions. To create each we need to access $Table_r$. On average the access is $O(1)$. There will be $m - 1$ accesses per expression. In total $O(m^2 * n)$ accesses. The priority queue operations (enqueue and dequeue) take $log(n)$ time, and the size of PQ is at most n. Therefore, the algorithm that finds the minimal expression for an instantiated b-expression is $O(c * (m^2 + log(n)) * n)$, where c is the number of the declarations in the b-expression. In general, for a b-expression that contains h $?_{head}$ symbols there can be n^h different instantiations. Thus, we conclude that the complexity of such an algorithm is $O(c * (m^2 + log(n)) * n^{h+1})$. Similarly, we show that the complexity of building $Table_r$ is $O((log(n)+m*n)*n)$. Therefore, the complexity of the entire algorithm is $O(c * (m^2 + log(n)) * n^{h+1} + (log(n) + m * n) * n)$. For an instantiate b-expression the algorithm is polynomial with the complexity $O(c * (m * (m + n) + log(n)) * n)$.

5.2 Loose Repair with N Minimal Expressions

We illustrate the algorithm that generate N expressions with the desired type τ that repair initial b-expression e_b, using declarations from the environment Γ (Figure 7). As before, we can split Γ into two b-environment Γ_b and r-environment Γ_r. We use function MinPath (Figure 5)to build and estimate the cost of the minimal paths between types and declarations. Also, we use declarations from Γ_b to mutate initial e_b, and this way potentially generate solutions that loosely follow the structure of e_b. We also argue that the algorithm is complete.

The algorithm is a modification of the A* search. Unlike regular A* our version operates on and builds partial expression. It starts from a hole $[e_b] : \tau$, containing an initial b-expression e_b and a desired type τ (step 3). It gradually unfolds the hole, creating new partial expressions, with new holes (the step 7). Every time we need to unfold a hole (substituted with a new partial expression) the algorithm try to embed the containing b-expression as much as possible (e.g. the first bullet in the first case). By embedding parts of the b-expression we follow its structure. This also reduces the size of the hole, i.e., its containing b-expression. Additionally, we replace the hole with r-declarations hoping the they might take us to the b-expression as well (the second bullet in the first case). Finally, we substitute the hole with b-declarations to mutate b-expression. This algorithm produces a set of mutated b-expressions allowing us to be complete. New b-expression is an approximation of the original b-expression.

To implement A* use the priority queue PQ for partial expressions. Each time a new partial expression is built we calculate its weight, in step 8, using function w as described in Section 4.2. The function w can be seen as a heuristic

LooseRepair(e_b, τ, Γ, N):
1. Using e_b find Γ_b, and then $\Gamma_r = \Gamma \setminus \Gamma_b$
2. We build an empty priority queue PQ that stores partial expressions.
   Smaller the cost of the partial expression, higher the priority in PQ.
3. We store hole $[e_b] : \tau$ in PQ.
4. If PQ is empty we terminate.
5. We remove first partial expression exp from PQ.
6. If exp is complete expression, has no holes, we put it in the Solutions set.
   If size of Solutions is N we terminate algorithm and output Solutions.
   Otherwise, we go to the step 4.
7. If exp has hole(s), choose one hole $h : \tau_h$ and:
   case $h = [f(e_1, \ldots, e_m)]$ and $f : \rho_1 \to \ldots \to \rho_m \to \rho \in \Gamma_b$:
   - If $\rho = \tau_h$ create a new partial expression exp by replacing $h : \tau_h$ with $f([e_1] : \rho_1, \ldots, [e_m] : \rho_m)$.
   - For each $g : \rho'_1 \to \ldots \to \rho'_m \to \tau_h \in \Gamma_r$ create m' different partial expressions: $g(h : \rho'_1, [?_{arg}] : \rho'_2, \ldots, [?_{arg}] : \rho'_m), \ldots, g([?_{arg}] : \rho'_1, \ldots, [?_{arg}] : \rho'_{m-1}, h : \rho'_m)$ to force the path to f through one of g's arguments. Using the m' partial expressions to replace $h : \tau_h$ in exp and create new m' expressions.
   - For each $d : \rho_1" \to \ldots \to \rho_m" \to \tau_h \in (\Gamma_b \setminus \{f : \rho_1 \to \ldots \to \rho_m \to \rho\}$ create a partial expression $d([?] : \rho_1", \ldots, [?] : \rho_m")$ and use it to replace $h : \tau_h$ in exp. This way we mutate the original b−expression.

   case $h = [?_{head}(e_1, \ldots, e_m)]$:
   - For each $g : \rho_1 \to \ldots \to \rho_m \to \tau_h \in \Gamma_r$ create a new partial expression exp by replacing $h : \tau_h$ with $g([e_1] : \rho_1, \ldots, [e_m] : \rho_m)$.
   - For each $g : \rho'_1 \to \ldots \to \rho'_m \to \tau_h \in \Gamma_r$ and $m \neq m'$ create m' different partial expressions: $g(h : \rho'_1, [?_{arg}] : \rho'_2, \ldots, [?_{arg}] : \rho'_m)$, ..., $g([?_{arg}] : \rho'_1, \ldots, [?_{arg}] : \rho'_{m-1}, h : \rho'_m)$ to force the path to f through one of g's arguments. Using the m' partial expressions to replace $h : \tau_h$ in exp and create new m' expressions.
   - For each $d : \rho_1" \to \ldots \to \rho_m" \to \tau_h \in \Gamma_b$ create a partial expression $d([?] : \rho_1", \ldots, [?] : \rho_m")$ and use it to replace $h : \tau_h$ in exp. This way we mutate the original b−expression.

   case $h = [?_{arg}]$:
   - For each $g : \rho_1 \to \ldots \to \rho_m \to \tau_h \in \Gamma_r$ create a new partial expression exp by replacing $h : \tau_h$ with $g([?_{arg}] : \rho_1, \ldots, [?_{arg}] : \rho_m)$.
   - For each $d : \rho_1" \to \ldots \to \rho_m" \to \tau_h \in \Gamma_b$ create a partial expression $d([?] : \rho_1", \ldots, [?] : \rho_m")$ and use it to replace $h : \tau_h$ in exp. This way we mutate the original b−expression.

   case $h = [?]$:
   for each $g : \rho_1 \to \ldots \to \rho_m \to \tau_h \in \Gamma$ create $g([?] : \rho_1, \ldots, [?] : \rho_m)$ partial expression and use it to replace $h : \tau_h$ in exp.
8. Calculate weights of the new partial expressions.
9. We put into PQ, the partial expressions that we encounter for the first time.
10. We go to the step 4.

Fig. 7. The algorithm that finds N expressions that repair as close as possible a given b-expression.

function that is a sum of the following two functions: (1) one that calculates weight of the constructed parts, and (2) one that calculates weight of all holes. The former relates to past cost function, and the latter to future cost function, of the A* heuristic function.

Theorem 1 (Completeness and Soundness). *Let Γ be a given environment, τ be a desired type, and e_b be a b-expression whose b-declarations are in Γ. Then it holds:*

$$\Gamma \vdash e : \tau \quad \textit{iff} \quad e \in \mathit{LooseRepair}(e_b, \tau, \Gamma, +\infty)$$

Proof:

If $\Gamma \vdash e : \tau$ then $e \in \mathit{LooseRepair}(e_b, \tau, \Gamma)$ direction: In the step 7, we substitute the hole $h : \tau$ with all declarations from (entire) Γ whose return type is τ. Although, in the different cases 6.x, we prioritize them differently based on the value of h. Eventually, all holes in each partial expression are substituted. Also, every unique partial expression is put and removed exactly once from PQ. This means we cover the entire search space.

If $e \in \mathit{LooseRepair}(e_b, \tau, \Gamma)$ then $\Gamma \vdash e : \tau$ direction: We start from a hole $h : \tau$, and continue producing only partial expression that return type τ. Note that we only combine declarations (in the step 7), such that they produce partial expressions that type check, i.e., return and argument types conform for every symbol and its subexpressions. Moreover, we only use Γ and holes to produce partial expressions. Finally, complete expressions will have no symbols $?$, $?_{arg}$, $?_{head}$ nor any hole. Thus they will be constructed only using Γ declarations. Therefore, we conclude that Γ derives an expression e with the type τ.

6 Related Work

We next provide an overview of related work on software repair and other approaches that share similar insights, ideas and techniques with our approach.

Type inference with automatic insertion of coercions. Several approaches on using coercions within type checking and type inference were presented in [13, 10]. This line of work focuses on type inference with automatic insertion of coercions in the context of functional programming languages. A recent work [19] presents a more sophisticated technique that extends Hindley-Milner type inference with coercive structural subtyping and goes beyond inserting coercions for local expression repairs in the type inference algorithm. The work presents an algorithm that derives subtype constraints from the whole target term, solves them to get a substitution consistent with the partial order on types, and finally applies the substitution to obtain a term that type-checks. Our approach conceptually differs from the aforementioned in multiple dimensions of the repair. The main difference is that our approach searches for any possible expression that is consistent with the types and structure as defined by the backbone expression. Additionally, our approach is limited to lambda calculus terms in the applicative long normal form and considers only base types without type constructors. The setup for the aforementioned approach requires mappings that

define coercions between types, while our approach leverages automatic search to find consistent function applications. While both approaches insert coercions only to function arguments, our approach considers all combinations of term applications that may produce coercion terms with appropriate types, together with any "boarding" arguments and mutations that may occur.

Automated inference of program fixes and contracts. These areas share the common goal of inferring code and rely on specialized search techniques [20, 20, 14]. Inferred software fixes and contracts are usually snippets of code that are synthesized according to the behavioral information gathered about the analyzed program. Such characterization is done by analyzing program state across the execution of tests; state can be defined using user-defined query operations [21, 20], and additional expressions extracted from the code [14]. Our code repair approach is restricted to applicative long normal form terms and considers only type information without characterization of program run-time behavior. Rather than filling of predefined code schemas at places in the existing code, our approach exhaustively searches through the space of all expressions that satisfy the constraint of a successful repair, while accounting for mutations of code.

Syntactic error diagnosis and repair A related line of work considers syntactic error diagnosis, as well as recovery and repair after such errors are found [4, 2]. The main target of these schemes is the parsing phase within the compilation process. Although our approach focuses on the repair of expressions that are correctly parsed according to the appropriate language grammar rules, some similarities may be noted. With respect to the these techniques and introduced taxonomy, our algorithm may be viewed as a type-checking repair scheme that is global, since it considers all the declarations in the scope for the repair, and interactive, since the developer can choose between multiple offered repair expressions and impact the repair process of subsequent errors. One of the most representative scheme for dealing with syntactic errors in LR and LL parsing is presented in [1]. Rather than just trying combinations of insertions, substitutions and deletions on a predefined length of the input after an error is detected, our approach considers the abstract syntax tree of the given expression and tries to encode all valid combinations of insertions in order to produce one or multiple expressions that would successfully type-check.

Searching for better type-error messages Interesting work, related to repairing code, has been done on improving type errors and corresponding error messages [12, 8, 9]. Although these approaches use techniques to modify type information [12], as well as the program [9], they target the problem of inappropriate type-checker's error messages for the purpose of giving better feedback to the developer. Our approach focuses on somewhat more ambitious goal of code repair and it does this by preserving the type-level information as well as the structure of ill-typed expressions. Some techniques used in these approaches are similar to the way backbone expressions are mutated in our approach, like argument addition and reordering [9]. The crucial difference is that such techniques are utilized only after an independent search mechanism determines places for

such modifications to be done, while our approach finds all suitable repair expressions according to the whole backbone expression and the weight heuristic.

Frameworks for deductive synthesis and execution of specifications. Frameworks that encompass verification, deductive synthesis and run-time constraint solving were presented in [6, 5]. Although these frameworks address more general goal of integrating software construction and verification while automating multiple aspects of the development process, in contrast to our approach that produces code fragments that do not need to satisfy formal specification, an interesting parallel can be drawn between such frameworks and our approach for code repair. When given a program that contains pieces of incomplete implementation, the aforementioned frameworks may employ techniques that synthesize missing fragments or allow solving them at run-time. Therefore, such approach may be viewed as repairing partial implementations that may happen at compile-time or may be deferred to run-time. While the domain of both includes only purely function programs, it is additionally constrained by the expressiveness of the underlying SMT solver theories, verification techniques, and deductive synthesis rules in the case of the frameworks, while restricted to (weak) applicative long-normal form in the case of our approach.

Sketching Program sketching tries to infer code fragments from the specification given as separate unoptimized programs [16, 17]. Practicality is achieved by focusing on particular domains that allow algorithms that employ a guided search over the syntax trees of the synthesized program with an a priori determined bound on the syntax tree size. In contrast, our repair approach requires merely a hint of the resulting expression and uses techniques to explore the unbounded space of expressions that are type-correct and conform to the structure of the backbone expression. Additionally, our approach is driven by externally defined heuristic measures to guide the search towards better solution candidates.

7 Conclusion

We described an approach that constructs well-typed expressions starting from ill-typed expressions and a set of type declarations. The algorithm can be applied in two ways: (1) to propose a ranked set of repaired expressions similar to the given ill-typed expression in an interactive manner, and (2) to automatically fix the given ill-typed expressions during the compilation process. We formalized the problem and proposed a calculus that synthesizes well-typed expressions based on a structure of the given expression with an arbitrary number of type-check errors. We described an algorithm that finds multiple repair expressions, and ranks them to allow getting the best-fitting one. The quality of returned solutions is measured by the similarity to the original given ill-typed expression and the system of weights. We showed that for a certain class of input expressions the algorithm is polynomial. Finally, we formulated theoretical guarantees of completeness and soundness for our algorithms.

References

1. M. G. Burke, G. A. Fisher, and T. J. Watson. A practical method for lr and ll syntactic error diagnosis and recovery. *ACM Transactions on Programming Languages and Systems*, 9:164–197, 1987.
2. P. Degano and C. Priami. Lr techniques for handling syntax errors. *Comput. Lang.*, 24(2):73–98, July 1998.
3. T. Gvero, V. Kuncak, I. Kuraj, and R. Piskac. Complete completion using types and weights. In *PLDI*, pages 27–38, 2013.
4. K. Hammond and V. J. Rayward-Smith. A survey on syntactic error recovery and repair. *Computer Languages*, 9(1):51–67, 1984.
5. E. Kneuss, V. Kuncak, I. Kuraj, and P. Suter. Synthesis modulo recursive functions. In *OOPSLA*, 2013.
6. V. Kuncak, E. Kneuss, and P. Suter. Executing specifications using synthesis and constraint solving (invited talk). In *Runtime Verification (RV)*, 2013.
7. C. Le Goues, W. Weimer, and S. Forrest. Representations and operators for improving evolutionary software repair. In *GECCO '12*, pages 959–966, 2012.
8. B. Lerner, D. Grossman, and C. Chambers. Seminal: searching for ml type-error messages. In *Proceedings of the 2006 workshop on ML*, ML '06, pages 63–73, 2006.
9. B. S. Lerner, M. Flower, D. Grossman, and C. Chambers. Searching for type-error messages. In *PLDI '07*, pages 425–434, 2007.
10. Z. Luo. Coercions in a polymorphic type system. *Mathematical. Structures in Comp. Sci.*, 18(4):729–751, Aug. 2008.
11. D. Mandelin, L. Xu, R. Bodík, and D. Kimelman. Jungloid mining: helping to navigate the api jungle. In *PLDI*, 2005.
12. B. J. McAdam. Repairing type errors in functional programs, 2001.
13. J. C. Mitchell. Coercion and type inference. In *POPL '84*, pages 175–185, 1984.
14. Y. Pei, Y. Wei, C. A. Furia, M. Nordio, and B. Meyer. Code-based automated program fixing. *ArXiv e-prints*, 2011. `arXiv:1102.1059`.
15. D. Perelman, S. Gulwani, T. Ball, and D. Grossman. Type-directed completion of partial expressions. In *PLDI*, pages 275–286, 2012.
16. A. Solar-Lezama, G. Arnold, L. Tancau, R. Bodik, V. Saraswat, and S. Seshia. Sketching stencils. In *PLDI '07*, pages 167–178, 2007.
17. A. Solar-Lezama, L. Tancau, R. Bodik, S. Seshia, and V. Saraswat. Combinatorial sketching for finite programs. *SIGARCH Comput. Archit. News*, 34(5):404–415, Oct. 2006.
18. V. Tannen, T. Coquand, C. A. Gunter, and A. Scedrov. Inheritance as implicit coercion. *Inf. Comput.*, 93(1):172–221, 1991.
19. D. Traytel, S. Berghofer, and T. Nipkow. Extending hindley-milner type inference with coercive structural subtyping. In *APLAS'11*, pages 89–104, 2011.
20. Y. Wei, C. A. Furia, N. Kazmin, and B. Meyer. Inferring better contracts. In *ICSE*, pages 191–200, 2011.
21. Y. Wei, Y. Pei, C. A. Furia, L. S. Silva, S. Buchholz, B. Meyer, and A. Zeller. Automated fixing of programs with contracts. In *ISSTA*, pages 61–72, 2010.