A. More Case Studies

In this section, we present eight additional case studies of TFix. TFix exactly matches the human fixes in the commits in four cases, removes the coding errors but does not perfectly match human fixes in two cases, and fails to generate a correct fix in the remaining two cases.

Exact matches The first case is shown in Figure 7, the developer forgot to write the `return` keyword in a getter function, causing a `getter-return` error in ESLint. As a result, the function does not return anything even though it can still run. TFix correctly inserts the missing `return`.

Figure 8 presents the second case. The developer declared variable `value` to be a constant type with the `const` keyword but then assigned a new value to it. This resulted in a `no-const-assign` error reported by ESLint. TFix changes `const` to `let`, successfully fixing the error. Note that the fix line is different from the error line in this example, showing the importance of the error context.

In Figure 9, the developer directly compared `counter` to `NaN`, which is a wrong way of checking if a variable is a valid value in JavaScript. The correct way is to use the library function `isNaN`, as suggested by TFix.

In the fourth case, as shown in Figure 10, the developer intended to check if `this.debug` is `true` but wrongly wrote an assignment in the if condition, which is a common programming mistake. Therefore, ESLint reported a `no-cond-assign` error. TFix fixes it by using `this.debug` solely as the condition.

The above four cases show that TFix is capable of generating human-like fixes for a wide range of errors.

Error removals Now we discuss two cases where TFix synthesizes fixes that correctly remove the errors but are different from human fixes. The first case is shown in Figure 11 where the developer used the `throw` statement on a string literal, causing a `no-throw-literal` error. Later, the developer fixed the error by printing the error message to the console. On the other hand, TFix proposes to construct and throw an `Error` instance, which is also correct.

The second case is shown in Figure 12 and the code contains a `guard-for-in` error for the same reason as discussed in Section 2. While the developer fixed the error by converting `for-in` to `forEach`, TFix proposes to add an if-check for the property. Both fixes remove the error correctly.

The above two cases show that TFix can generate correct fixes that are syntactically different from but functionally the same as human fixes.

Failures Next, we describe two cases where TFix could not generate a correct fix. In the first case, shown in Figure 13, the programmer assigned a value to itself, which does not make sense from a programming perspective. ESLint denotes such errors with `no-self-assign`. In the commit, the variable was assigned to the correct variable, namely `modified_`. TFix failed to generate a correct fix due to its fixed windows size, i.e., the variable `modified_` is not in the error context. Increasing the window size can potentially resolve this failure.

The second case is shown in Figure 14. The erroneous code contains an empty `catch` block, and the developer fixed it by simply printing the caught error to console. Unfortunately, TFix generates a fix identical to the incorrect code and is unable to solve the problem. The reason might be that similar samples do not often occur in our fine-tuning dataset. Enlarging the fine-tuning dataset can help TFix handle more errors.

```
get [Symbol.iterator]() {
    this.__data__[Symbol.iterator]();
}
```

```
get [Symbol.iterator]() {
    return this.__data__[Symbol.iterator]();
}
```

Figure 7. TFix fixes a `getter-return` error by inserting the missing return keyword.

```
const { value ='' } = e.detail || {};
value = this.getValue(newValue);
this._trigger('onChange', { detail: {value } });
```

```
let { value ='' } = e.detail || {};
value = this.getValue(newValue);
this._trigger('onChange', { detail: {value } });
```

Figure 8. TFix fixes a `no-const-assign` error by changing the variable's declaration type.

```
var randomCounter = false;
if (counter == NaN) {
    counter = Math.floor(Math.random * 10000);
```

```
var randomCounter = false;
if (isNaN(counter)) {
    counter = Math.floor(Math.random * 10000);
```

Figure 9. TFix fixes a `use-isnan` error by using the correct function supported by JavaScript.

```
var signal = JSON.parse(message.content);
if(this.debug = true)
    console.log(message);
```

```
var signal = JSON.parse(message.content)
if(this.debug)
    console.log(message);
```

Figure 10. TFix fixes a `no-cond-assign` error by correcting the condition.

Coding error:

```
if (destination == null) {
    throw "destination is invalid";
}
```

Human fix:

```
if (destination == null) {
    console.log("destination is invalid");
}
```

Output of TFix:

```
if (destination == null) {
    throw new Error("destination is invalid");
}
```

Figure 11. TFix correctly fixes a `no-throw-literal` error by constructing an `Error` instance while the human developer prints the error message to console.

Coding error:

```
applyCss(obj) {
  for (var key in obj) {
        this.element.style[key] = obj[key];
```

Human fix:

```
applyCss(obj) {
    Object.keys(obj).forEach(key) => {
        this.element.style[key] = obj[key];
```

Output of TFix:

```
applyCss(obj) {
    for (var key in obj) {
        if (!obj.elements.hasOwnProperty(key)) continue;
        this.element.style[key] = obj[key];
```

Figure 12. TFix correctly fixes a `guard-for-in` error by adding an if-check while the human developer uses `forEach`.

Coding error:

```
{
    modified = modified;
}
```

Human fix:

```
{
    modified = modified_;
}
```

Output of TFix:

```
{
    this.modified = parseInt(false);
}
```

Figure 13. TFix generates irrelevant code as a fix.

Coding error:

```
} catch (e) {

}
```

Human fix:

```
} catch (e) {
    log.warn(error);
}
```

Output of TFix:

```
} catch (e) {

}
```

Figure 14. TFix generates a fix identical to the erroneous code.

B. Running Data Extraction on an Example

In this section, we explain our data extraction pipeline in Algorithm 1 in more detail by running it on a simple, synthesized commit. The process is similar in principle but more complicated in detail for real-world commits.

Example commit with error fixing Figure 15 shows our example commit (p, p'). From p to p', the commit removes Line 6, adds Line 7, and adds a new `return` token at Line 9. First, TFix runs `detector` to detect errors on p and p' (Line 3 of Algorithm 1). For p, `detector` detects a set of three errors $\mathcal{E} = \{e_1, e_2, e_3\}$:

e_1: `no-this-before-super` at Line 3.

e_2: `getter-return` at Line 9.

e_3: `no-throw-literal` at Line 13.

e_2 is fixed in the commit with the newly added `return` token. Therefore, for p', `detector` identifies two errors $\mathcal{E} = \{e'_1, e'_2\}$:

e'_1: `no-this-before-super` at Line 3.

e'_2: `no-throw-literal` at Line 13.

Since $|\mathcal{E}| > |\mathcal{E}'|$ (Line 4 of Algorithm 1), we proceed the extraction procedure as the commit contains an error fix.

Finding fixed errors with bipartite matching Next TFix calls the `findFixedErrors` function to identify the set of errors $\mathcal{E}_{\text{fixed}} \subseteq \mathcal{E}$ fixed in the commit (Line 5 of Algorithm 1). To achieve this, `findFixedErrors` first invokes the greedy bipartite matching procedure between $\mathcal{E}$ and $\mathcal{E}'$ to find the set of unfixed errors $\mathcal{E}_{\text{unfixed}} \subseteq \mathcal{E}$. We iterate all pairs of errors in $\mathcal{E}$ and $\mathcal{E}'$ to see if they are the same error. Clearly, $e_1 = e'_1$ and $e_3 = e'_2$. Therefore, $\mathcal{E}_{\text{unfixed}} = \{e_1, e_3\}$ and $\mathcal{E}_{\text{fixed}} = \mathcal{E} - \mathcal{E}_{\text{unfixed}} = \{e_2\}$.

Computing target fix Then we compute the fix for e_2 with the `computeFix` function (Line 7 of Algorithm 1). We first leverage the Myers diff algorithm to obtain a series of three edit operations:

1. Delete a whole line (Line 6).
2. Insert a new line (Line 7).
3. Insert `return` at Line 9 after the second tab.

We perform the above edit operations starting from p and track how the error line l_k of e_2 (Line 9) shifts with each edit operation to obtain the target fix line $l_{k'}$. The deletion

```
1   class HumanPlayer extends Player {
2       constructor(name) {
3           this.name = name;
4           this.health = 100;
5       }
6   -   set health (health) { this.health = health; }
7   +   get health () { return this.health; }
8       get name() {
9           +return this.name;
10      }
11      damage(x) {
12          if (x < 0) {
13              throw "No negative damage";
14          }
15          this.health -= x;
16      }
17  }
```

Figure 15. An example commit fixing a `getter-return` error.

at Line 6 pulls l_k up by one line while the insertion at Line 7 pushes it down again, so we compute that $l_{k'}$ stays at Line 9. In the end, we obtain the following error context $\mathcal{L}_k$ and fix context $\mathcal{L}_{k'}$ as a sample in our dataset:

```
    get name() {
        this.name;
    }
```

```
    get name() {
        return this.name;
    }
```