

Высокоуровневые представления изменений в коде на основе AST не зависящие от языка

Дмитрий Абрамов

Научный руководитель: Тимофей Брыксин

ВШЭ, Санкт-Петербург

11 Февраля 2020

- Код в процессе разработки претерпевает изменения
- Изменения нужно представлять в каком-то виде
- Представления изменений используемые в системах контроля версий требуют быстроедействие, легковесности и универсальности
- Такие представления отделены от структуры кода
- Для исследования эволюции ПО и задач машинного обучения для программной инженерии (ML4SE) хочется иметь представления изменений, которые отражают структуру кода и лучше демонстрируют намерения разработчика

Построчное сравнение

Редакционный сценарий — набор удаленных и добавленных строк.

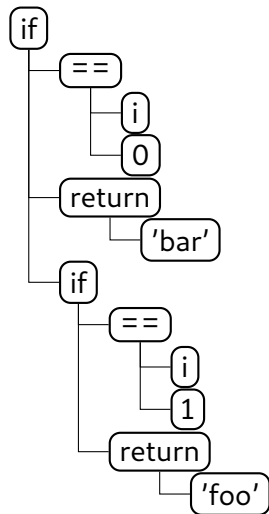
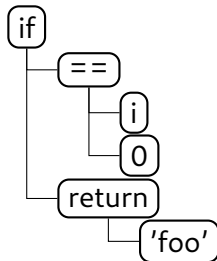
- + Производительный: алгоритм Мейерса — $\mathcal{O}(nd)$ с использованием линейной памяти [Myers, 1986]
- + Универсальный, широко используется во многих приложениях: системы контроля версий, Unix diff и т. д.
- Строки не соответствуют сложной структуре кода, грубое представление
- Далеко от реальных изменений произведенных разработчиком

```
-      execute(taskDecorator(decorate, command));  
+      int decorated = taskDecorator(decorate, command);  
+  
+      if (decorated != command) {  
+          decoratedTaskMap(put, decorated, command);  
+      }  
+  
+      execute(decorated);
```

Абстрактные Синтаксические Деревья (AST)

```
if (i == 0) {  
    return "foo";  
}
```

```
if (i == 0) {  
    return "bar";  
} else if (i == 1) {  
    return "foo";  
}
```



Соответствие (Matching)

множество M пар $m \in \text{«вершины AST до»} \times \text{«вершины AST после»}$,
которые соответствуют одному и тому же элементу программы

Минимальный согласованный редакционный сценарий относительно соответствия M

— кратчайший редакционный сценарий, который не содержит
действий «удалить, добавить» с вершинами, входящими в
соответствие M .

Алгоритм Chawathe [Chawathe et al., 1996].

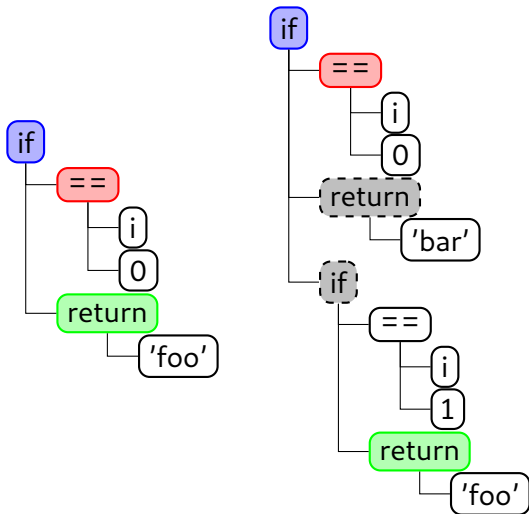
- Найти соответствие M общих вершин в обоих деревьях
- Построить редакционный сценарий для соответствия M
- Вся сложность кроется в первом шаге

Сравнение AST — Соответствие

Одинаковым цветом обозначены пары вершин из соответствия.

```
if (i == 0) {  
    return "foo";  
}
```

```
if (i == 0) {  
    return "bar";  
} else if (i == 1) {  
    return "foo";  
}
```



GumTree — инструмент для построения редакционных сценариев, состоящих из действий «удалить, добавить, изменить, переместить» [Falleri et al., 2014].

- Свой алгоритм матчинга — два обхода дерева
- Набор эвристик, метрики близости поддеревьев
- Текущий state-of-the-art в сравнении AST — это небольшие улучшения GumTree:
 - Удаление поддеревьев которых не коснулись изменения [Yang and Whitehead, 2019]
 - Добавление действия Copy-Paste в редакционный сценарий [Higo et al., 2017]
- Не зависит от языка программирования

Редакционный сценарий — набор действий с вершинами синтаксического дерева.

- + Изменения полностью отражают иерархическую структуру кода
- Генерация оптимальных редакционных сценариев — сложно:
 - С действиями «удалить, добавить, изменить» — $\mathcal{O}(n^3)$ [Pawlik and Augsten, 2011]
 - Добавление действия «переместить» делает задачу NP-трудной [Bille, 2005]
- Изменения слишком низкоуровневые, чтобы отражать реальные намерения программиста и разумные манипуляции кодом

Рефакторинг

типичное изменение кода не меняющее результата его исполнения, имеющее описанное название и структуру.

Текущий state-of-the-art — **Refactoring Miner** — инструмент для выделения рефакторингов из истории git репозитория, точность более 90% Tsantalis et al. [2018].

- + Рефакторинг — сложное изменение совершаемое разработчиком намеренно
- Рефакторинги — это очень малая часть всех изменений, которые происходят с кодом

Другие подходы

Глубокое обучение

- Обучение векторных представлений (embedding)
- + Гибкий метод
- Отсутствие интерпретируемости
- Большая ошибка

Суммирование изменений

[Liu et al., 2019; Hu et al., 2018; Shido et al., 2019]

- Представление изменений в виде описания на естественном языке
- + Помогает понимать изменения человеку, полезно для code-review
- Слишком высокий уровень абстракции, теряется много информации; возможны ошибки

Другие подходы — Высокоуровневые изменения

- Change Distiller — инструмент выделяющий 50 типов Java-специфичных изменений из разницы между AST. [Fluri et al., 2007]
 - CLDiff - используется похожий подход для получения высокоуровневых изменений в Java коде, чтобы затем искать описанные авторами взаимосвязи между изменениями. [Huang et al., 2018]
- + Высокоуровневые изменения содержащие информацию о структуре и в то же время отражают изменения на уровне близком к тому, на котором оперируют разработчики пишущие код
- Типы изменений сконструированы вручную специально для языка Java

Цель

Разработать не зависящий от языка программирования инструмент для представления изменений в программном коде отражающих его структуру на более высоком уровне абстракции, чем действия с отдельными вершинами AST.

Задачи:

- Разработать не зависящий от языка алгоритм построения высокоуровневых редакционных сценариев
- Создать расширяемый инструмент для получения diff-а на основе GumTree и нового алгоритма
- Оценить разработанный подход на датасете изменений языка Java и Python и сравнить с существующими Java-специфичными подходами
- Проверить разработанный подход, применив его к задачам ML4SE

Высокоуровневое представление изменений кода

Требуемые характеристики:

- Высокоуровневое представление без потери полной информации о структуре
- Возможность выразить любое изменение
- Минимум зависимости от языка

Решение

- Используем низкоуровневые редакционные сценарии содержащие действия над вершинами AST.
- Затем объединим эти действия в группы

Зависимость от языка I

- Совсем отказаться от информации о языке программирования сложно
- Можно взять только то, что важно
- Важна роль вершины в AST

```
1  foo(x);  
2  
1  if (!cached.contains(x)) {  
2      int value = foo(x);  
3      cached.put(x, value);  
4  }
```

- Синтаксические категории делятся на:
 - операции (statements)
 - выражения (expressions)
- Операции в свою очередь делим на:
 - атомарные (нет потомков-операций в AST)
 - вложенные (есть потомки-операции в AST)
- Опираясь на эти параметры, будем объединять действия из низкоуровневого редакционного сценария

Алгоритм получения высокоуровневых изменений

- Алгоритм в качестве входных данных использует:
 - AST версии «до» и версии «после»
 - Соответствие (Matching) вершин AST двух версий
 - Редакционный сценарий, состоящий из 4 типов действий:
 - удалить (del)
 - добавить (add)
 - изменить (upd)
 - переместить (mov)
 - Действие имеет вид: *имя(Вершина, Новый Родитель, Позиция)*
 - Например, действие `add(n1, n9, 3)` добавляет вершину `n1` третьим потомком к вершине `n9`
- Алгоритм, Шаг 1: перемещение вершины — само по себе высокоуровневое действие
 - Для каждого действия `mov(X, Y, i)` добавляем высокоуровневое действие `moveNodeX(X, Y, i)`

Алгоритм — Пример

Рассмотрим алгоритм на примере изменения из коммита 3c1adf7 из репозитория spring-framework.

```
public void execute(Runnable command) {  
    super.execute(taskDecorator.decorate(command));  
    Runnable decorated = taskDecorator.decorate(command);  
    if (decorated != command) {  
        decoratedTaskMap.put(decorated, command);  
    }  
    super.execute(decorated);  
}
```

Шаг 1 — Пример

```
| -Block (n1)  
  | -ExpressionStatement (n2)  
    | -SuperMethodInvocation (n3)  
      | -SimpleName:execute (n4)  
      | -MethodInvocation (n5)  
        | -SimpleName:taskDecorator (n6)  
        | -SimpleName:decorate (n7)  
        | -SimpleName:command (n8)
```

add(n10, n1, 1)	add(n21, n20, 1)
add(n19, n1, 2)	add(n22, n20, 2)
add(n11, n10, 1)	add(n24, n23, 1)
add(n13, n10, 2)	add(n33, n3, 2)
add(n20, n19, 1)	add(n25, n24, 1)
add(n23, n19, 2)	add(n26, n25, 1)
add(n12, n11, 1)	add(n27, n25, 2)
add(n14, n13, 1)	add(n28, n25, 3)
mov(n5, n13, 2)	add(n29, n25, 4)

moveMethodInvocation(n5, n13, 2)

```
| -Block (n9)  
  | -VariableDeclarationStatement (n10)  
    | | -SimpleType:Runnable (n11)  
    | | | -SimpleName:Runnable (n12)  
    | | -VariableDeclarationFragment (n13)  
    | | | -SimpleName:decorated (n14)  
    | | -MethodInvocation (n15)  
    | | | -SimpleName:taskDecorator (n16)  
    | | | -SimpleName:decorate (n17)  
    | | | -SimpleName:command (n18)  
  | -IfStatement (n19)  
    | | -InfixExpression:!= (n20)  
    | | | -SimpleName:decorated (n21)  
    | | | -SimpleName:command (n22)  
    | | -Block (n23)  
    | | | -ExpressionStatement (n24)  
    | | | | -MethodInvocation (n25)  
    | | | | | -SimpleName:decTaskMap (n26)  
    | | | | | -SimpleName:put (n27)  
    | | | | | -SimpleName:decorated (n28)  
    | | | | | -SimpleName:command (n29)  
  | -ExpressionStatement (n30)  
    | -SuperMethodInvocation (n31)  
      | -SimpleName:execute (n32)  
      | -SimpleName:decorated (n33)
```

- Цель второго шага — сгруппировать действия удаления и добавления вершин, относящиеся к операциям
- Действие удаления вершины аналогично действию добавления в обратном изменении: алгоритмы для обработки добавлений и удалений симметричны.
- Для каждого действия A , добавляющего вершину N рассмотрим потомков N
- Каждая вершина-потомок x может быть двух типов:
 - *Основная* — для вложенных операций это сама x , её потомки-выражения на глубине 1, потомки-блоки; для атомарных операций – это сама x вместе с потомками
 - *Составляющая* — есть только у вложенных операций, это все потомки-операции x на глубине 1 и потомки-блоки на глубине 2

Шаг 2 — Алгоритм

ACTIONS — множество действий.

Для каждого действия $A \in \text{ACTIONS}$, $A = \text{AddNode } N$:

```
1  BASE = {}; COMP = {}
2  for statement node x in N.descendants:
3      A = x.action
4      if A is base action:
5          add A to BASE
6      else:
7          add A to COMP
8
9  if  $\exists A \in \text{BASE}$  and A.node is in some matching M:
10     isPartial = true
11  else:
12     if  $\nexists A \in \text{COMP}$  and A.node is in some matching M:
13         remove COMP from ACTIONS
14  remove BASE from ACTIONS
15  return HighLevelAdd(N, isPartial)
```

Шаг 2 — Пример I

```
| -Block (n1)  
  | -ExpressionStatement (n2)  
    | -SuperMethodInvocation (n3)  
      | -SimpleName:execute (n4)  
      | -MethodInvocation (n5)  
        | -SimpleName:taskDecorator (n6)  
        | -SimpleName:decorate (n7)  
        | -SimpleName:command (n8)
```

add(n10,n1,1)	add(n21,n20,1)
add(n19,n1,2)	add(n22,n20,2)
add(n11,n10,1)	add(n24,n23,1)
add(n13,n10,2)	add(n33,n3,2)
add(n20,n19,1)	add(n25,n24,1)
add(n23,n19,2)	add(n26,n25,1)
add(n12,n11,1)	add(n27,n25,2)
add(n14,n13,1)	add(n28,n25,3)
mov(n5,n13,2)	add(n29,n25,4)

addVarDeclPARTIAL(n10,n1,1)

moveMethodInvocation(n5,n13,2)

```
| -Block (n9)  
  | -VariableDeclarationStatement (n10)  
    | | -SimpleType:Runnable (n11)  
    | | | -SimpleName:Runnable (n12)  
    | | -VariableDeclarationFragment (n13)  
    | | | -SimpleName:decorated (n14)  
    | | -MethodInvocation (n15)  
    | | | -SimpleName:taskDecorator (n16)  
    | | | -SimpleName:decorate (n17)  
    | | | -SimpleName:command (n18)  
  | -IfStatement (n19)  
    | | -InfixExpression:!= (n20)  
    | | | -SimpleName:decorated (n21)  
    | | | -SimpleName:command (n22)  
    | | -Block (n23)  
      | -ExpressionStatement (n24)  
        | -MethodInvocation (n25)  
          | -SimpleName:decTaskMap (n26)  
          | -SimpleName:put (n27)  
          | -SimpleName:decorated (n28)  
          | -SimpleName:command (n29)  
      | -ExpressionStatement (n30)  
        | -SuperMethodInvocation (n31)  
          | -SimpleName:execute (n32)  
          | -SimpleName:decorated (n33)
```

Шаг 2 — Пример II

```
| -Block (n1)  
  | -ExpressionStatement (n2)  
    | -SuperMethodInvocation (n3)  
      | -SimpleName:execute (n4)  
      | -MethodInvocation (n5)  
        | -SimpleName:taskDecorator (n6)  
        | -SimpleName:decorate (n7)  
        | -SimpleName:command (n8)
```

add(n10,n1,1)	add(n21,n20,1)
add(n19,n1,2)	add(n22,n20,2)
add(n11,n10,1)	add(n24,n23,1)
add(n13,n10,2)	add(n33,n3,2)
add(n20,n19,1)	add(n25,n24,1)
add(n23,n19,2)	add(n26,n25,1)
add(n12,n11,1)	add(n27,n25,2)
add(n14,n13,1)	add(n28,n25,3)
mov(n5,n13,2)	add(n29,n25,4)

```
addVarDeclPARTIAL(n10,n1,1)  
addIfStatement(n19,n1,2)
```

```
moveMethodInvocation(n5,n13,2)
```

```
| -Block (n9)  
  | -VariableDeclarationStatement (n10)  
    | | -SimpleType:Runnable (n11)  
    | | | -SimpleName:Runnable (n12)  
    | | -VariableDeclarationFragment (n13)  
    | | | -SimpleName:decorated (n14)  
    | | -MethodInvocation (n15)  
    | | | -SimpleName:taskDecorator (n16)  
    | | | -SimpleName:decorate (n17)  
    | | | -SimpleName:command (n18)  
  | -IfStatement (n19)  
    | | -InfixExpression:!= (n20)  
    | | | -SimpleName:decorated (n21)  
    | | | -SimpleName:command (n22)  
    | | -Block (n23)  
    | | | -ExpressionStatement (n24)  
    | | | | -MethodInvocation (n25)  
    | | | | | -SimpleName:decTaskMap (n26)  
    | | | | | -SimpleName:put (n27)  
    | | | | | -SimpleName:decorated (n28)  
    | | | | | -SimpleName:command (n29)  
    | | -ExpressionStatement (n30)  
    | | | -SuperMethodInvocation (n31)  
    | | | | -SimpleName:execute (n32)  
    | | | | -SimpleName:decorated (n33)
```

- Остались только действия с выражениями и действия изменения вершины
- Для каждого действия A с вершиной n находим предка x , который является операцией
- Добавляем A в список L_x
- Для каждого x генерируем высокоуровневое действие
Update x with L_x

Шаг 3 — Пример

```
| -Block (n1)  
  | -ExpressionStatement (n2)  
    | -SuperMethodInvocation (n3)  
      | -SimpleName:execute (n4)  
      | -MethodInvocation (n5)  
        | -SimpleName:taskDecorator (n6)  
        | -SimpleName:decorate (n7)  
        | -SimpleName:command (n8)
```

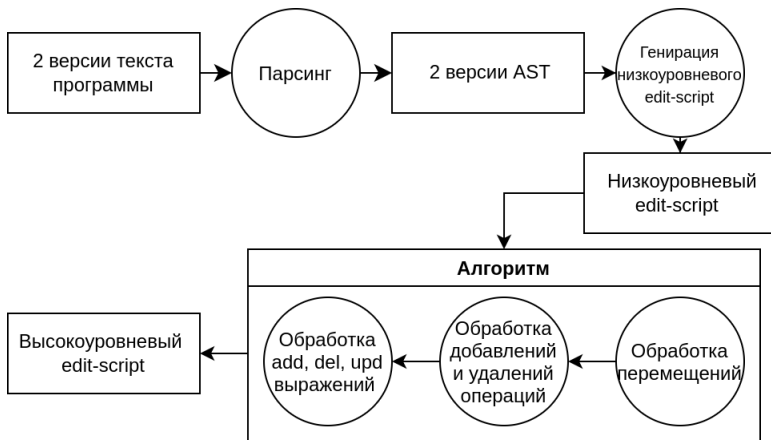
add(n10,n1,1)	add(n21,n20,1)
add(n19,n1,2)	add(n22,n20,2)
add(n11,n10,1)	add(n24,n23,1)
add(n13,n10,2)	add(n33,n3,2)
add(n20,n19,1)	add(n25,n24,1)
add(n23,n19,2)	add(n26,n25,1)
add(n12,n11,1)	add(n27,n25,2)
add(n14,n13,1)	add(n28,n25,3)
mov(n5,n13,2)	add(n29,n25,4)

```
addVarDeclPARTIAL(n10,n1,1)  
addIfStatement(n19,n1,2)  
updateExpressionStatement(n2) with  
  addSimpleName(n33,n3,2)  
moveMethodInvocation(n5,n13,2)
```

```
| -Block (n9)  
  | -VariableDeclarationStatement (n10)  
    | | -SimpleType:Runnable (n11)  
    | | | -SimpleName:Runnable (n12)  
    | | -VariableDeclarationFragment (n13)  
    | | | -SimpleName:decorated (n14)  
    | | -MethodInvocation (n15)  
    | | | -SimpleName:taskDecorator (n16)  
    | | | -SimpleName:decorate (n17)  
    | | | -SimpleName:command (n18)  
  | -IfStatement (n19)  
    | | -InfixExpression:!= (n20)  
    | | | -SimpleName:decorated (n21)  
    | | | -SimpleName:command (n22)  
    | | -Block (n23)  
    | | | -ExpressionStatement (n24)  
    | | | | -MethodInvocation (n25)  
    | | | | | -SimpleName:decTaskMap (n26)  
    | | | | | -SimpleName:put (n27)  
    | | | | | -SimpleName:decorated (n28)  
    | | | | | -SimpleName:command (n29)  
  | -ExpressionStatement (n30)  
    | -SuperMethodInvocation (n31)  
      | -SimpleName:execute (n32)  
      | -SimpleName:decorated (n33)
```


- Полученный алгоритм генерирует редакционные сценарии, удовлетворяющие требованиям корректности и сохранения информации о структуре
- Разделение синтаксических категорий по типам — несложный процесс, это единственная зависимость от языка
- Основная идея алгоритма — получать достаточно высокоуровневые изменения, при этом разделяя принципиально разные действия

- Для генерации низкоуровневых редакционных сценариев используется GumTree.
 - Один из state-of-the-art подходов
 - Позволяет поддержать новый ЯП
 - Самый используемый в сообществе
- Реализация на языке Kotlin, использование кода GumTree напрямую



Чтобы поддержать новый язык программирования необходимо:

- Реализовать обертку над парсером, которая преобразует результат парсинга в AST в терминах GumTree — дерево из вершин, хранящих информацию о типе, текстовом вхождении и представлении
- Разметить типы вершин в файле конфигурации языка (вложенные и атомарные операции, выражения, блоки).
- Алгоритм использует информацию вершинах, являющихся блоками операций
- Нужно преобразовать AST полученное парсингом, чтобы оно содержало явные вершины типа «блок операций», если такие элементы не предусмотрены парсером (пример — парсер Python)

Поддержка языка — Пример

```
1 {  
2     "language_id": "Python",  
3     "extensions": [".py"],  
4     "atomic_statements": [  
5         12, 32, 33 ...  
6     ],  
7     "nested_statements": [  
8         15, 41 ...  
9     ],  
10    "block_statements": [42]  
11 }
```

Цели оценки:

- Сравнить высокоуровневые изменения получаемые представленным инструментом с Java-специфичными подходами
 - CLDiff [Huang et al., 2018]
 - ChangeDistiller [Fluri et al., 2007]
- Оценить высокоуровневые изменения полученные для отличающегося от Java языка — Python
- Сравнить характеристики GumTree и HLDiff — производительность, размер редакционных сценариев

Сравнение с Java-специфичными подходами

Наборы данных:

- CVSVintage - история репозиториях крупных Java-проектов
- Майнинг данных с GitHub

Вопросы:

По шкале { *хуже; немного хуже; одинаково; лучше* }

- Как HLDiff отражает намерения разработчика по сравнению с другим инструментом?
- Как HLDiff помогает понять произошедшие с кодом изменения по сравнению с другим инструментом?

Оценка изменений в Python-коде

Поддержка Python в HLDiff:

- Обертка для парсера на основе ASDL или модуля ast
- Преобразование AST: добавление вершин-блоков для блоков операций
- Версии Python3

Данные: 150k Python Dataset, GitHub

Вопрос:

- Каков процент удачно выделенных высокоуровневых изменений - элементов высокоуровневого редакционного сценария, которые помогают понять произошедшие с кодом изменения лучше, чем низкоуровневый редакционный сценарий.

Мотивация

- Более объективная оценка предложенного подхода к генерации высокоуровневых изменений
- Высокоуровневые изменения могут улучшить существующие подходы, так как потенциально содержат в себе более точную информацию об изменениях и об их совместном использовании
- Аналогичный переход к более высокоуровневым структурам сильно улучшил результаты в области обработки естественного языка

- Получить представление (embedding) высокоуровневого редакционного сценария
- Обучить модель с использованием обоих типов редакционных сценариев
- Обучить модель с использованием высокоуровневого редакционного сценария

Результаты

- Создан алгоритм для генерации высокоуровневых изменений, сохраняющий информацию об иерархической структуре кода и использующий минимум информации о языке программирования
- Реализован инструмент на основе данного алгоритма; предусмотрена возможность расширения на новые языки программирования

Планы

- Оценка генерируемых представлений изменений, сравнение с Java-специфичными подходами, оценка на изменениях в коде на Python
- Применение полученного подхода для улучшения существующей модели ML4SE, работающий с изменениями

- Bille, P. (2005). A survey on tree edit distance and related problems. *Theor. Comput. Sci.*, 337(1-3):217–239.
- Chawathe, S. S., Rajaraman, A., Garcia-molina, H., and Widom, J. (1996). Change detection in hierarchically structured information. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 493–504.
- Falleri, J., Morandat, F., Blanc, X., Martinez, M., and Monperrus, M. (2014). Fine-grained and accurate source code differencing. In *ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014*, pages 313–324.
- Fluri, B., Wuersch, M., Plnitzer, M., and Gall, H. (2007). Change distilling: Tree differencing for fine-grained source code change extraction. *IEEE Trans. Softw. Eng.*, 33(11):725–743.

- Higo, Y., Ohtani, A., and Kusumoto, S. (2017). Generating simpler ast edit scripts by considering copy-and-paste. In *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017*, pages 532–542, Piscataway, NJ, USA. IEEE Press.
- Hu, X., Li, G., Xia, X., Lo, D., Lu, S., and Jin, Z. (2018). Summarizing source code with transferred api knowledge. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI-18*, pages 2269–2275. International Joint Conferences on Artificial Intelligence Organization.
- Huang, K., Chen, B., Peng, X., Zhou, D., Wang, Y., Liu, Y., and Zhao, W. (2018). Cldiff: Generating concise linked code differences. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018*, pages 679–690, New York, NY, USA. ACM.

- Liu, Q., Liu, Z., Zhu, H., Fan, H., Du, B., and Qian, Y. (2019). Generating commit messages from diffs using pointer-generator network. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pages 299–309.
- Myers, E. W. (1986). An $O(nd)$ difference algorithm and its variations. *Algorithmica*, 1(2):251–266.
- Pawlik, M. and Augsten, N. (2011). Rted: A robust algorithm for the tree edit distance. *Proceedings of the VLDB Endowment (PVLDB)*, 5:334.
- Shido, Y., Kobayashi, Y., Yamamoto, A., Miyamoto, A., and Matsumura, T. (2019). Automatic source code summarization with extended tree-lstm. *arXiv preprint arXiv:1906.08094*.

- Tsantalis, N., Mansouri, M., Eshkevari, L. M., Mazinianian, D., and Dig, D. (2018). Accurate and efficient refactoring detection in commit history. In *Proceedings of the 40th International Conference on Software Engineering, ICSE '18*, page 483–494, New York, NY, USA. Association for Computing Machinery.
- Yang, C. and Whitehead, E. (2019). Pruning the ast with hunks to speed up tree differencing. pages 15–25.