High-level Language-independent AST-based Source Code Differencing

Dmitrii Abramov

Scientific Advisor: Timofei Bryksin

National Research University Higher School of Economics, St Petersburg

March 23, 2020

Outline

Introduction

- Source code is constantly changing
- These changes require proper representation
- Line-based differencing is good for Version Control Systems
- Such representations are coarse-grained and do not reflect the code structure
- Machine Learning For Software Engineering tasks and Code Review process may benefit from more comprehensive and structure-aware representations

Line-based differencing

Edit-script

a representation of changes as a set of actions transforming one version to another.

- Edit script — set of actions *remove line* and *add line*

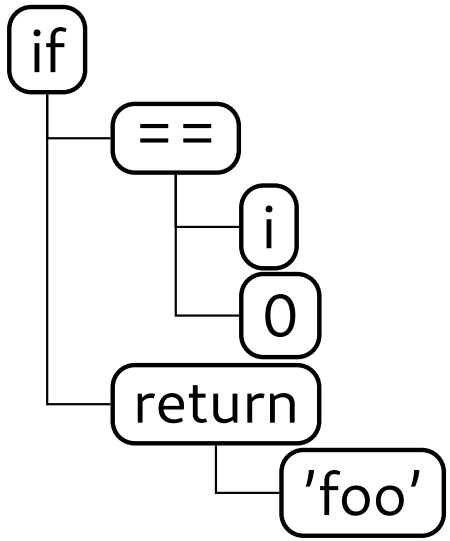
+ Universal, applicable to any text file
+ Performant — Meyers algorithm [Myers, 1986]

– Does not reflect the code structure

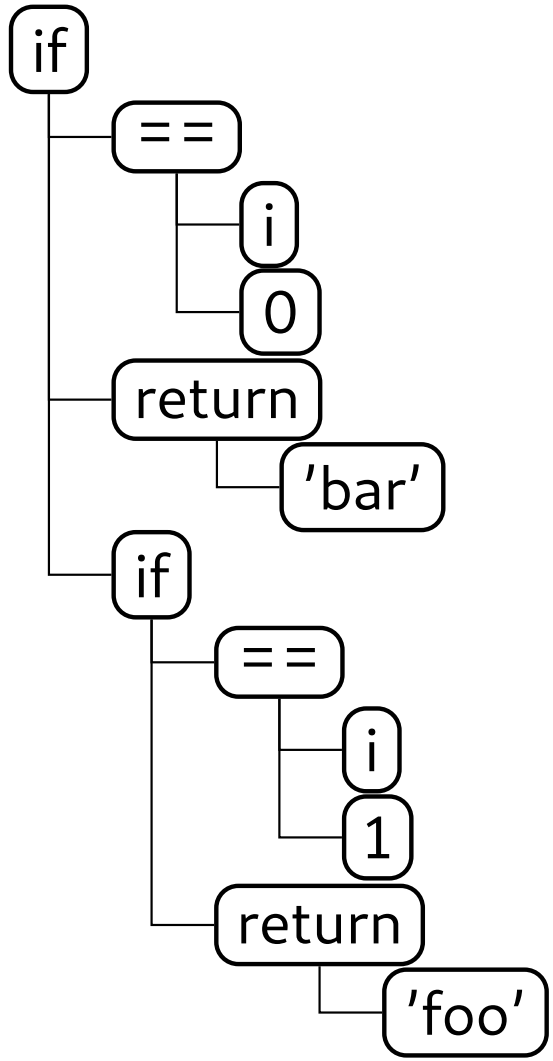
```
-         execute(taskDecorator(decorate, command));
+         int decorated = taskDecorator(decorate, command);
+
+         if (decorated != command) {
+             decoratedTaskMap(put, decorated, command);
+         }
+
+         execute(decorated);
```

Abstract Syntax Tree (AST)

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




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

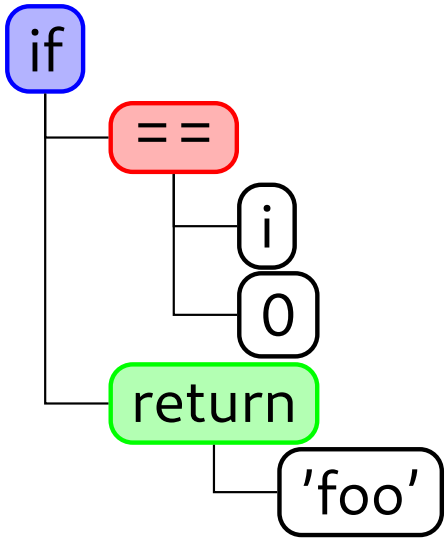


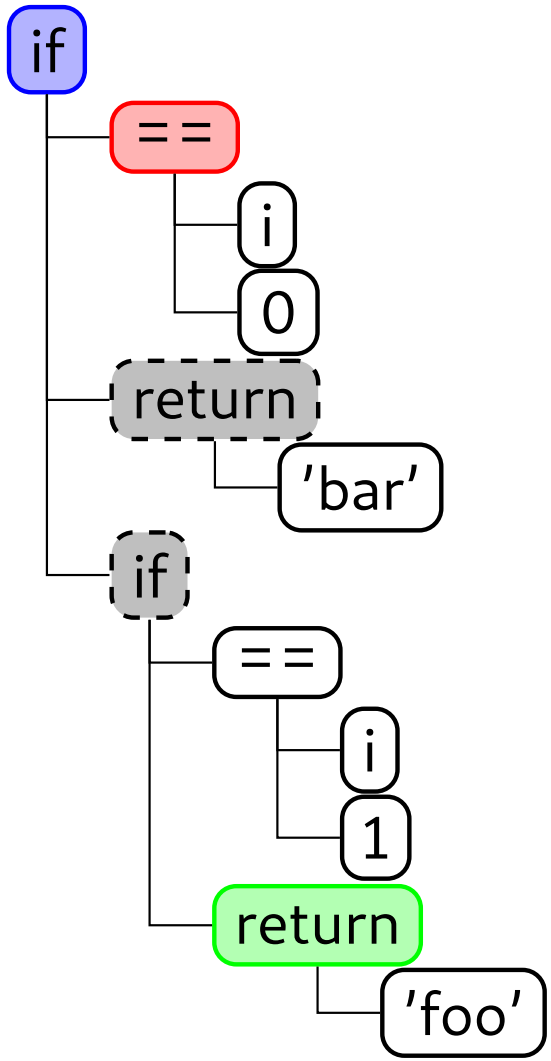
Comparing ASTs — Matching

Одинаковым цветом обозначены пары вершин из соответствия.

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

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





Comparing ASTs

Edit Script — set of actions with tree nodes.

State-of-the-art approach: GumTree [Falleri et al., 2014] and it's enhancements [Yang and Whitehead, 2019; Higo et al., 2017].

+ Reflect hierarchical structure of the code
− Generating optimal edit-scrips is a hard problem
  - Actions «*add, remove, update*» — $\mathcal{O}(n^3)$ time complexity [Pawlik and Augsten, 2011]
  - Considering the action «*move*» makes the problem NP-hard [Bille, 2005]
− Representation is too low-level to reflect the correlation between changes to nodes

Another approaches

- Refactoring detection
  - \+ Reflects developer's intention
  - \+ Good Results — more than 90% precision [Tsantalis et al., 2018]
  - − Very small amount of changes are refactorings
- General Deep Learning Approaches
  - \+ Flexible approach
  - − Non-interpretable
  - − High error
- Summarising Code Changes
  - \+ Natural language representation
  - − Too many information lost
  - − High error

High-level AST differencing

- Change Distiller — detects 50 types of Java-specific changes to ASTs.
  [Fluri et al., 2007]
- CLDiff - similiar approach to high-level changes detection to ASTs.
  [Huang et al., 2018]

+ High-level changes aware of the code structure and reflecting correlation between changes
− The core idea behind approach is Java-specific, which is a serious threat to validity
− Language specific approach limits possible academic and practical applications

Method

Goal

Create a tool for generating high-level, AST-based, and language-independent code changes representations.

Tasks:

- Design an algorithm for high-level edit scripts generation
- Create extensible tool based on the algorithm and the GumTree
- Evaluate this approach on Java and Python code changes and compare with existing Java-specific works
- Test the approach applicability to ML4SE problems

High-level changes representation

Требуемые характеристики:

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

Решение

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

Language information dependency I

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

```
1  foo(x);
```

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

Language information dependency II

- 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 репозиория 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(n19,n1,2)
add(n11,n10,1)
add(n13,n10,2)
add(n20,n19,1)
add(n23,n19,2)
add(n12,n11,1)
add(n14,n13,1)
mov(n5,n13,2)
```

```
add(n21,n20,1)
add(n22,n20,2)
add(n24,n23,1)
add(n33,n3,2)
add(n25,n24,1)
add(n26,n25,1)
add(n27,n25,2)
add(n28,n25,3)
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)
```

Шаг 2

- 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 $A \in$ ACTIONS, A = 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 ∃ A ∈ BASE and A.node is in some matching M:
10     isPartial = true
11 else:
12     if ∄ A ∈ 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)
```

```
|-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)
```

add(n10,n1,1)
add(n19,n1,2)
add(n11,n10,1)
add(n13,n10,2)
add(n20,n19,1)
add(n23,n19,2)
add(n12,n11,1)
add(n14,n13,1)
mov(n5,n13,2)

add(n21,n20,1)
add(n22,n20,2)
add(n24,n23,1)
add(n33,n3,2)
add(n25,n24,1)
add(n26,n25,1)
add(n27,n25,2)
add(n28,n25,3)
add(n29,n25,4)

---

addVarDeclPARTIAL(n10,n1,1)

moveMethodInvocation(n5,n13,2)

Шаг 2 — Пример II

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

```
|-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)
```

add(n10,n1,1)
add(n19,n1,2)
add(n11,n10,1)
add(n13,n10,2)
add(n20,n19,1)
add(n23,n19,2)
add(n12,n11,1)
add(n14,n13,1)
mov(n5,n13,2)

add(n21,n20,1)
add(n22,n20,2)
add(n24,n23,1)
add(n33,n3,2)
add(n25,n24,1)
add(n26,n25,1)
add(n27,n25,2)
add(n28,n25,3)
add(n29,n25,4)

---

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

moveMethodInvocation(n5,n13,2)

Шаг 3 — Алгоритм

- 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(n19,n1,2)
add(n11,n10,1)
add(n13,n10,2)
add(n20,n19,1)
add(n23,n19,2)
add(n12,n11,1)
add(n14,n13,1)
mov(n5,n13,2)
```

```
add(n21,n20,1)
add(n22,n20,2)
add(n24,n23,1)
add(n33,n3,2)
add(n25,n24,1)
add(n26,n25,1)
add(n27,n25,2)
add(n28,n25,3)
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)
```

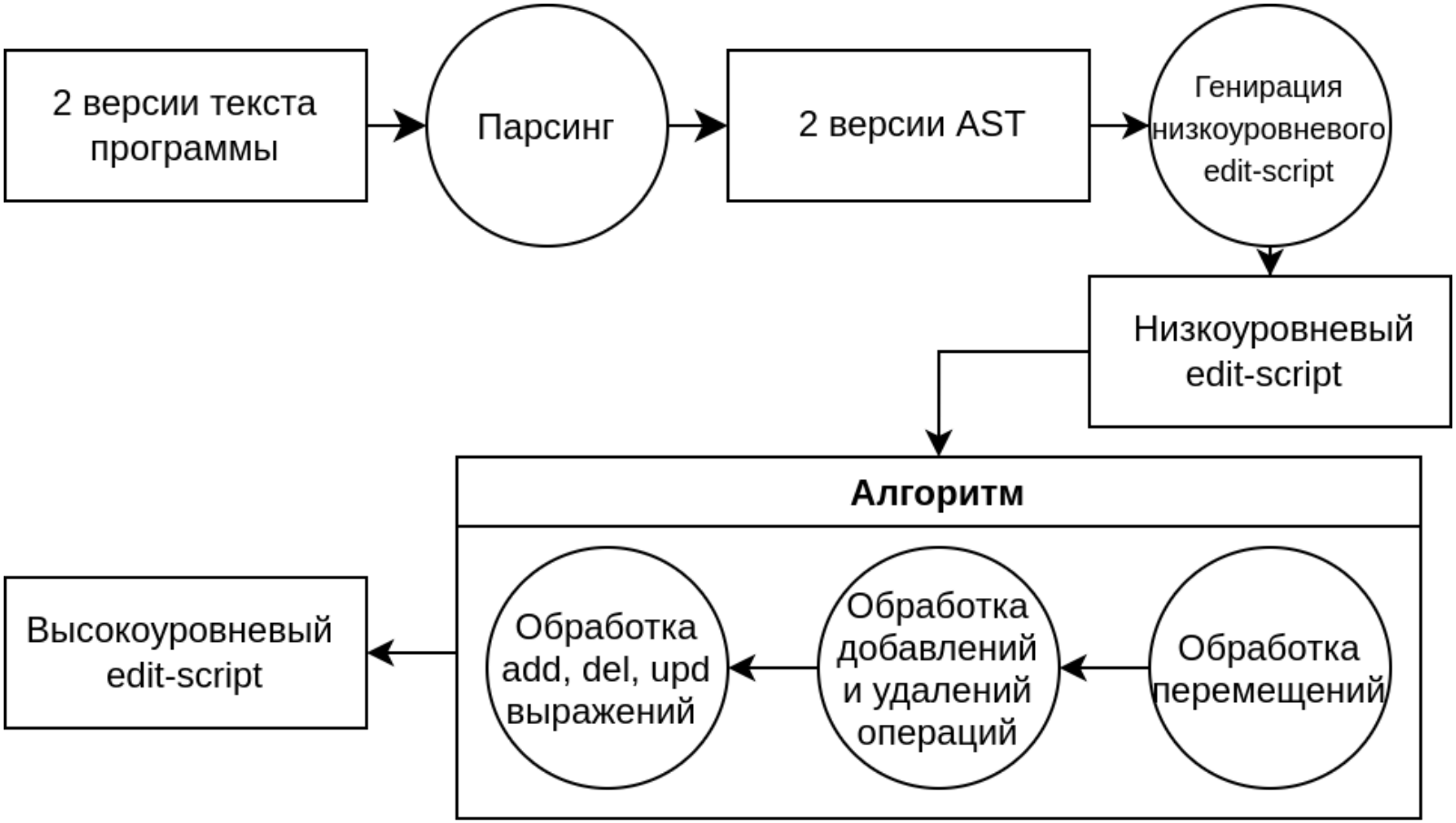
Алгоритм — вывод

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

Инструмент HLDiff

- 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

Архитектура



Инструмент HLDiff — поддержка нового ЯП

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 }
```

Оценка полученных высокоуровневых изменений

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 репозиториев 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 высокоуровнего редакционного сценария, которые помогают понять произошедшие с кодом изменения лучше, чем низкоуровневый редакционный сценарий.

Применение подхода к задаче ML4SE

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

Применение подхода к задаче ML4SE – метод

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

Вывод

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

References I

Bille, P. (2005). A survey on tree edit distance and related problems. *Theor. Comput. Sci.*, 337(1-3):217–239.

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., PInzger, 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.

References II

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.

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.

Tsantalis, N., Mansouri, M., Eshkevari, L. M., Mazinanian, 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.

References III

Yang, C. and Whitehead, E. (2019). Pruning the ast with hunks to speed up tree differencing. pages 15–25.