Разработка инструмента для сбора обогащенных AST

Автор: Е.С. Спирин, ММО201С
Научный руководитель: к.т.н., доцент Т.А. Брыксин

Машинное обучение для кода

Появляется все больше работ по применению ML к программному коду – [1], [2], [3]

Способы представления кода в моделях

- ➔ Текстовое представление
  Токены коды представляют собой слова естественного языка
- ➔ Абстрактные синтаксические деревья (ASTs)
  Используют структурные особенности кода
  - ◆ ASTs и его производные превосходят простые подходы – [4], [5], [6], [7]

[1] – Ferreira et al., “Software engineering meets deep learning: A mapping study”, 2020
[2] – Li et al., “Deep Learning & Software Engineering: State of Research and Future Directions”, 2019
[3] – Yang et al., “A Survey on Deep Learning for Software Engineering”, 2020
[4] – Mou et al., “Convolutional Neural Networks over Tree Structures for Programming Language Processing”, 2016
[5] – Zhang et al., “Retrieval-based neural source code summarization”, 2020
[6] – Alon et al., “A General Path-Based Representation for Predicting Program Properties”, 2019
[7] – Allamanis et al., “Learning to Represent Programs with Graphs”, 2018

IntelliJ Platform и PSI деревья

jetbrains.com/opensource/idea/

Открытая платформа для разработки IDEs (IntelliJ IDEA, Android Studio, ...)

- ➔ Предоставляет функциональность по **работе с кодом**
  - Переименование сущностей
  - Перемещение фрагментов кода
  - Переход к местам объявлениям или разрешение ссылок
  - Автодополнение кода
  - ...
- ➔ Использует Program Structure Interface (PSI) для **представление кода**
  - Хранит в себе конкретное синтаксическое дерево (CST) программы
  - Предоставляет функции для работы с деревом

Цель и задачи

Цель: разработка инструмента по извлечению и обработке PSI деревьев из IntelliJ Platform.

Задачи:

1. провести обзор предметной области;
   a. изучить способы представления кода в моделях машинного обучения;
   b. изучить способы работы с кодом в IntelliJ Platform;
2. спроектировать и разработать инструмент по извлечению и обработке PSI деревьев из IntelliJ Platform;
3. провести апробацию разработанного инструмента.

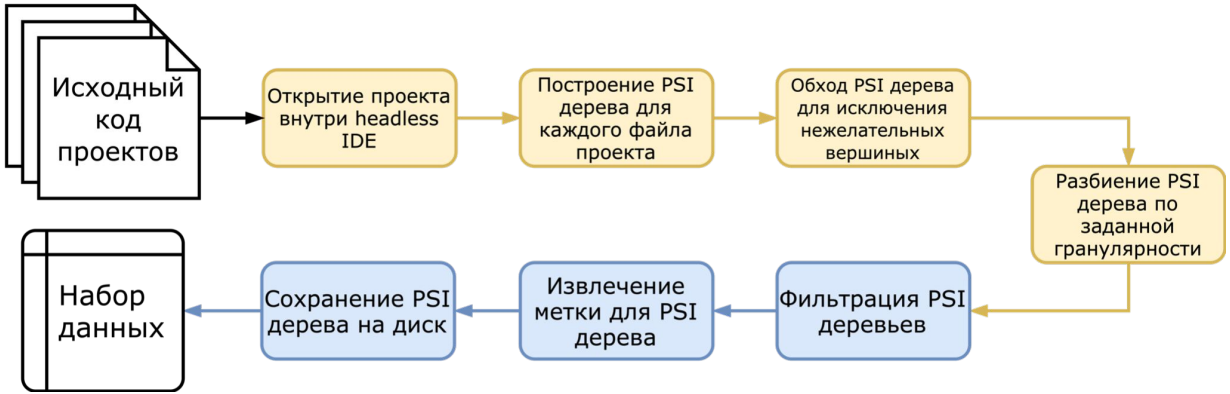
PSIMiner – главное

- ➔ Использует алгоритмы статического анализа платформы IntelliJ Platform для извлечения обогащенных представлений кода
- ➔ Строит готовый к обучению набор данных по исходному коду
  - ◆ Уже содержит популярные сценарии подготовки данных
  - ◆ Расширяется для большего количества сценариев использования с помощью готовых интерфейсов
- ➔ Поддерживает Java и Kotlin
  - ◆ PSI доступен для множества популярных языков
  - ◆ Гибкая архитектура позволяет добавлять новые языки

JetBrains-Research/psiminer

PSIMiner – технические детали

- ➔ PSIMiner – плагин к IntelliJ IDEA, работающей в headless режиме
- ➔ Две логические части:
  - Извлечение и обработка PSI деревьев
  - Создание набора данных под нужную задачу и модель
- ➔ В любой момент можно добавить логику по извлечению новой информации

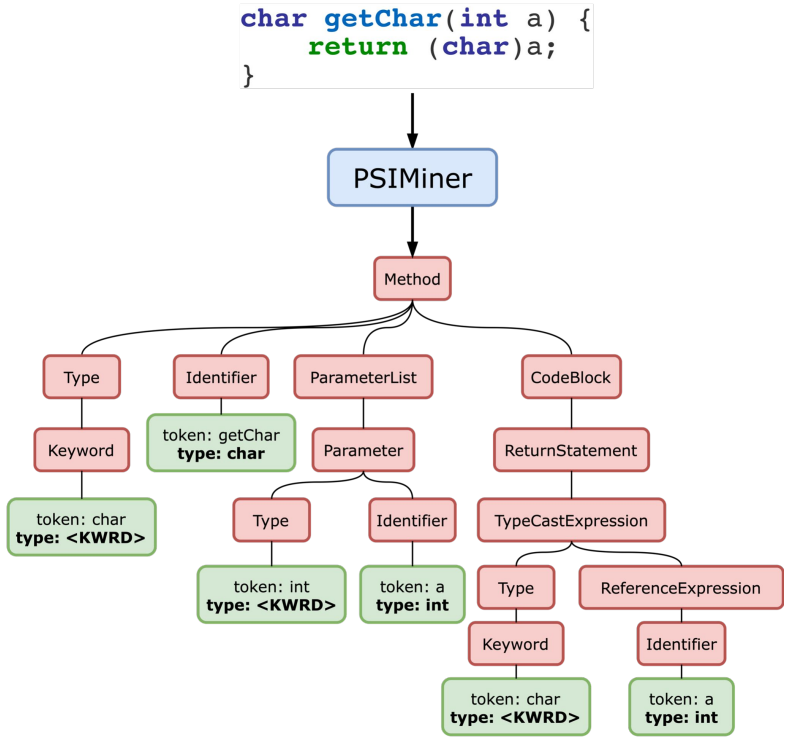


PSIMiner – пример обогащения

Возможное применение

➔ Добавление информации о типах в листья AST

- Если лист соответствует переменной – добавляем ее тип
- Если лист соответствует вызову функции – добавляем тип возвращаемого значения
- Специальные токены для листьев с операторами, ключевыми словами.
- ...

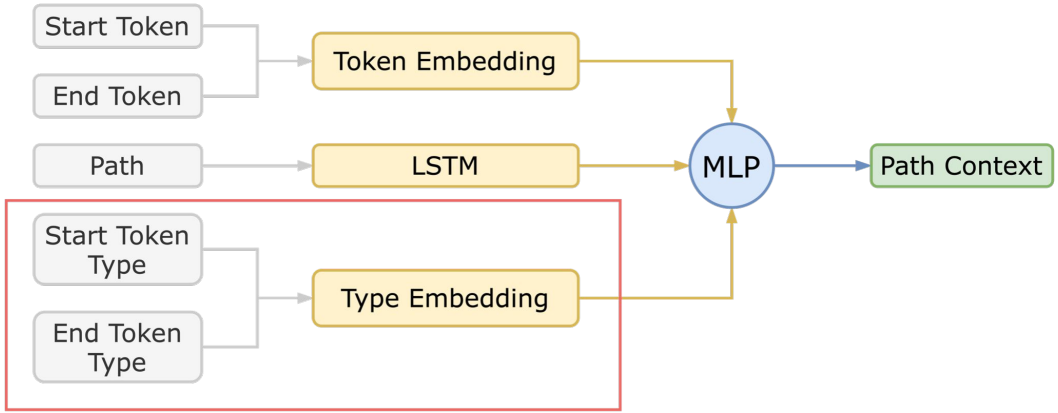


Модификация моделей

Code2seq

Alon et al. “Code2seq: Generating Sequences from Structured Representations of Code”

- ➔ AST представляется в виде набора путей – последовательность вершин между двумя листьями
- ➔ Кодер строит векторное представление каждого пути



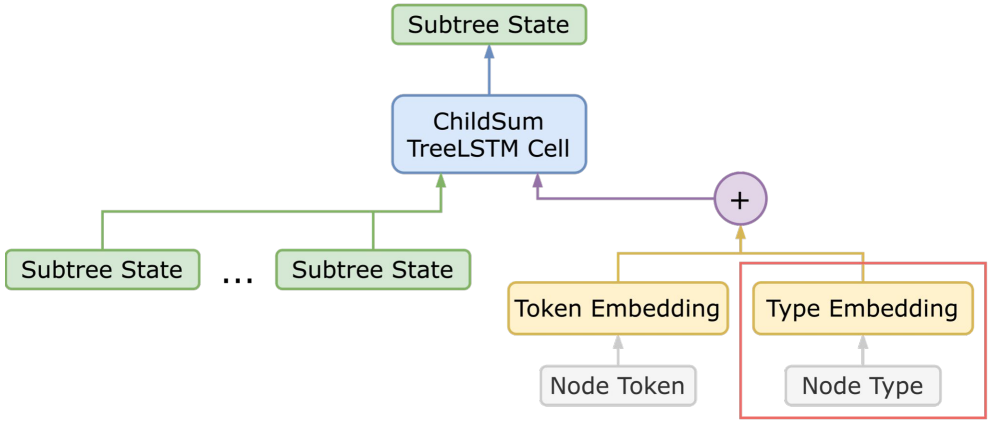
JetBrains-Research/code2seq

Модификация моделей

TreeLSTM

Tai et al. “Improved Semantic Representations From Tree-Structured Long Short-Term Memory Networks”

- ➔ Адаптация ячейки LSTM для работы с деревьями
- ➔ Ячейка строит векторное представление поддерева по состояниям дочерних вершин и информации в текущей вершине



JetBrains-Research/
embeddings-for-trees

Результаты модификаций

Сравнение оригинальной и расширенной типами code2seq моделей на задаче предсказания имени метода

Dataset	Model	Precision	Recall	F1 score
Java-small	Original	38.76	28.63	32.93
	Type-enhanced	**41.68**	**29.76**	**34.73**
Java-med	Original	**51.38**	39.08	44.39
	Type-enhanced	49.15	**42.35**	**45.5**

Сравнение оригинальной и расширенной типами TreeLSTM моделей на задаче предсказания имени метода

Dataset	Model	Precision	Recall	F1 score
Java-small	Original	35.39	23.59	28.31
	Type-enhanced	**36.06**	**25.13**	**29.62**

Результаты

1. Проведен обзор предметной области. Изучены способы представления кода в ML-моделях и способы работы с кодом в IntelliJ Platform.
2. Спроектирован и разработан PSIMiner, инструмент для извлечения и обработки PSI деревьев из IntelliJ Platform.
3. Проведена апробация инструмента. С помощью PSIMiner в AST была добавлена информация о типах листовых вершин, что повысило качество ML-моделей.

Результаты работы частично представлены на конференции MSR'21 (Paper, Demo)

JetBrains-Research/psiminer

JetBrains-Research/code2seq

JetBrains-Research/embeddings-for-trees

Неопубликованные результаты

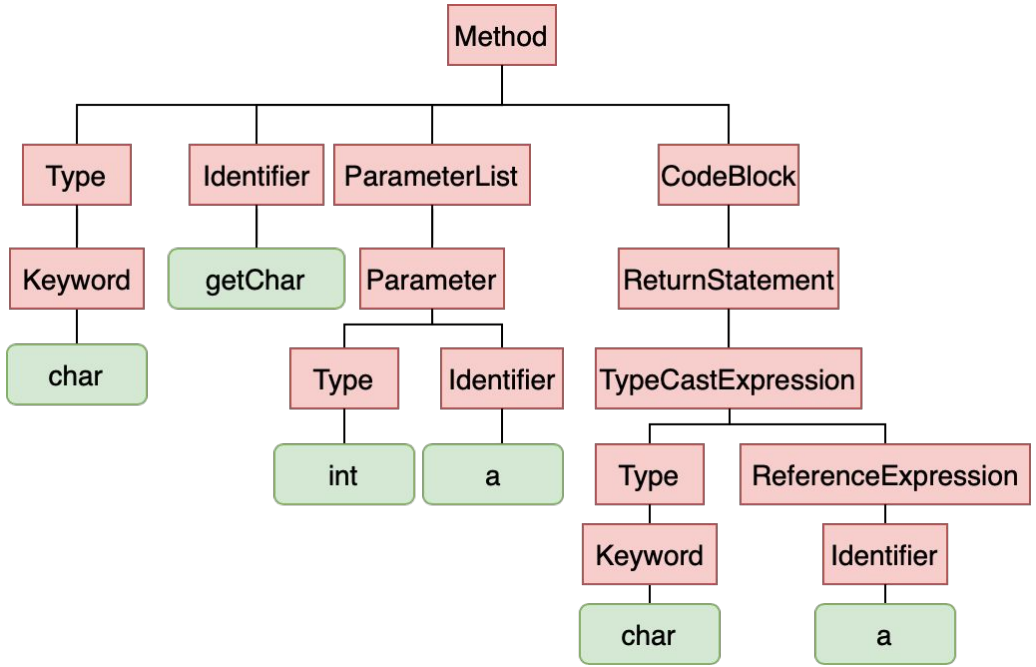
- ➔ Поддержка нескольких языков
  - PSI деревья поддерживают множество языков
  - PSIMiner предоставляет ряд интерфейсов по добавлению новых языков
  - Добавлена поддержка Kotlin
- ➔ Разделение инструмента на Core и CLI
  - Часть логики PSIMiner направлена на запуск headless IntelliJ IDEA и обработку входных параметров
  - Теперь основную логику легче переиспользовать в сторонних проектах
- ➔ Дополнительный эксперимент с добавлением типов
  - Модифицирована модель TreeLSTM для работы с типами

Абстрактное синтаксическое дерево

- ➔ Представление результата синтаксического анализа программы в виде дерева
  - Внутренняя вершина – применение продукции грамматики
    - Какая операция выполняется над детьми
  - Листовая вершина – терминал
    - Операнд (токен) – переменная, литерал, другая функция и т.п.
- ➔ Результат работы синтаксического анализа компилятора
- ➔ Исследуются на протяжении очень долгого времени

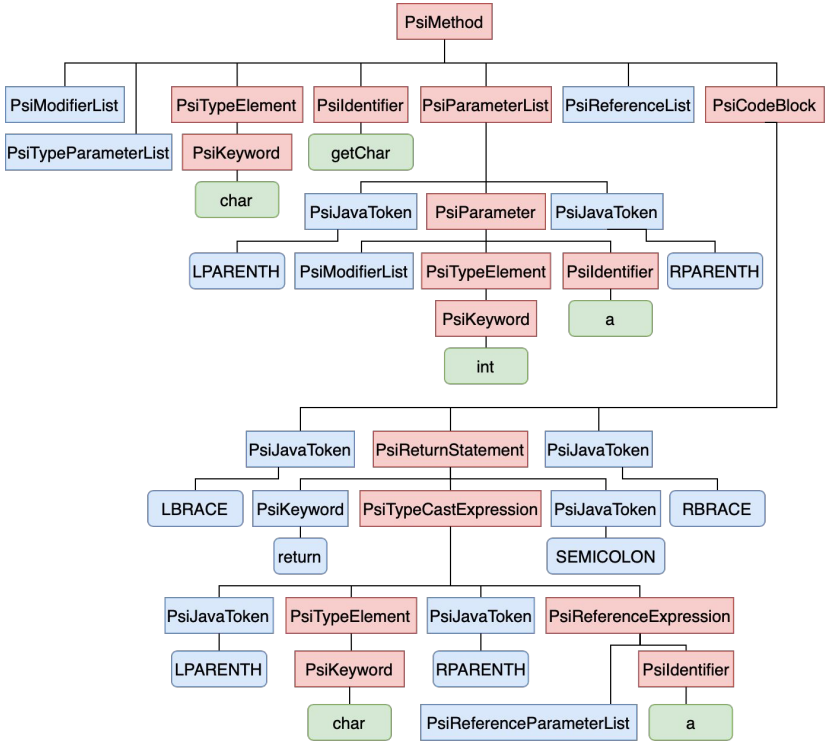
Абстрактное синтаксическое дерево

```
char getChar(int a) {
    return (char)a;
}
```



PSI дерево

```
char getChar(int a) {
    return (char)a;
}
```



Пример конфигурации

```
{
  "filters": [
    {"name": "constructor"},
    {"name": "empty method"},
    {
      "name": "by modifiers",
      "excludeModifiers": ["abstract"]
    }
  ],
  "label": {"name": "method name prediction"},
  "storage": {"name": "json typed tree"},

  "languages": ["Java", "Kotlin"],
  "node ignore rules": [
    {"name": "whitespace"},
    {"name": "empty grammar lists"},
    {"name": "keywords"},
    {"name": "java symbols"}
  ],
  "tree transformations": [
    {"name": "compress operators"},
    {
      "name": "hide literals",
      "hideNumbers": true
    },
    {
      "name": "remove comments",
      "removeJavaDoc": false
    }
  ],

  "printTrees": false
}
```

Выделение нужной гранулярности

- ➔ Единицей набора данных в коде могут выступать
  - Метод
    Method name prediction, move method refactoring, ...
  - Класс
    Class name prediction, ...
  - Файлы
    Классификация алгоритмов (POJ-104), поиск дубликатов, ...
- ➔ Пользователь сам задаёт необходимый уровень гранулярности
- ➔ PSIMiner разбивает исходные PSI на нужный уровень
  - Поддерживается функция, класс и файл.

Фильтрация AST

Garbage in – garbage out

```
protected abstract fun isGoodTree(
    root: PsiElement,
    language: Language
): Boolean
```

➔ Удаление конструкторов даёт улучшение на задачи предсказания имён функций

➔ Поддерживаются

- Фильтр функций по модификаторам, аннотациям, наличию тела
- Фильтр на основе числа строк кода
- Фильтр на основе размера дерева

Выделение меток

Часто используется self-supervised подход при обучении моделей

```
protected abstract fun handleTree(
    root: PsiElement,
    language: Language
): String?
```

➔ Метка для каждого примера извлекается из самого примера
- Имя функции
- JavaDoc или docstring

➔ Извлечение метки требует и обработки примера чтобы избежать утечку в данных
- Если используем имя функции, то надо убедиться, что в коде/AST не осталось этого имени – маскирование

Сохранение на диск

Разные модели ⇒ разный формат входных данных (пути, деревья, графы...)

```
protected abstract fun convert(
    labeledTree: LabeledTree,
    outputDirection: OutputDirection
): String
```

Поддерживается

- ➔ Code2seq storage

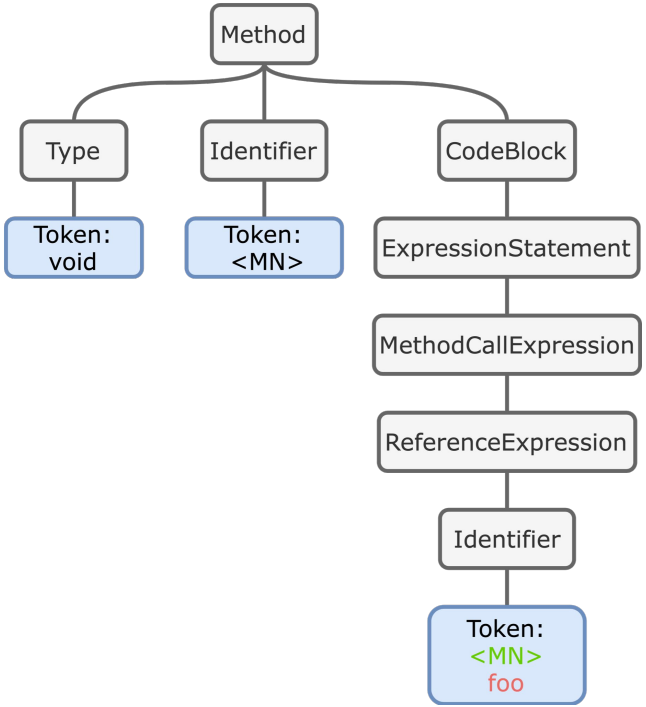
```
get|type|signature json|value,3|2,<MN> type|signature,4|2,<MN> json|value,3|4,type|signature
```

- ➔ JSON AST storage (Python150K)

```
{"label": "get|type|signature",
 "AST":
     [{"node":"METHOD","children":[1,2,3],"token":"<EMPTY>"},
      {"node":"MODIFIER_LIST|ANNOTATION|JAVA_CODE_REFERENCE|IDENTIFIER","token":"json|value"},
      {"node":"TYPE|JAVA_CODE_REFERENCE|IDENTIFIER","token":"type|signature"},
      {"node":"IDENTIFIER","token":"<MN>"}]}
```

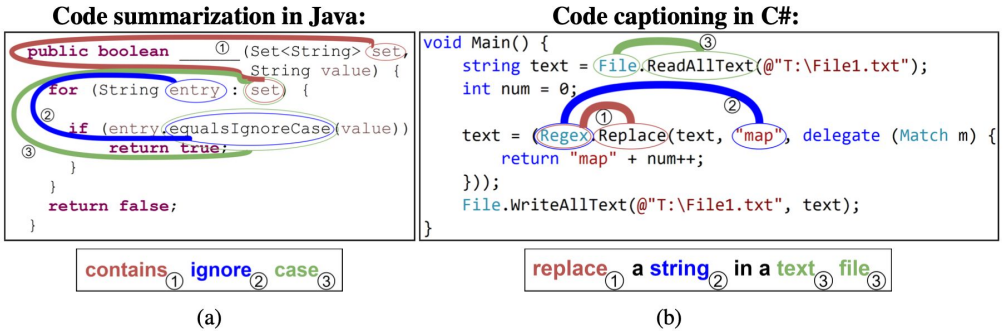
Утечка в данных

- ➔ Результаты ванильного code2seq значительно ниже, чем в оригинальной работе на тех же данных
- ➔ Нашли утечку в данных – в оригинальной работе не маскировались рекурсивные вызовы функций
- ➔ Повторение утечки поднимает “качество” до заявленных чисел



Code2seq

[Alon et al. “Code2seq: Generating Sequences from Structured Representations of Code”]

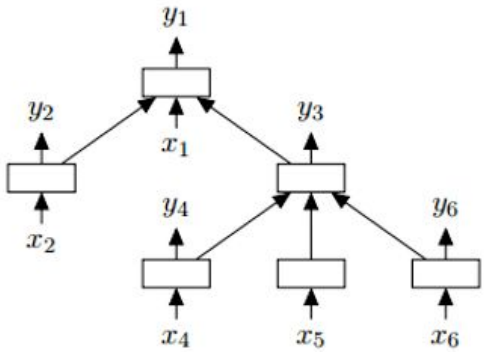


1. Token embedding
   a. Токен разбивается на подтокены
      `mySuperVariable → [my, super, variable]`
   b. Вектор токена – сумма векторов подтокенов
2. Path embedding
   a. Вектор пути – результат применения bi-LSTM

$$z = tanh\left(W_{in}\left[encode\_path(v_1 ... v_l); encode\_token(value(v_1)); encode\_token(value(v_l))\right]\right)$$

Tree-LSTM

Tai et al. “Improved Semantic Representations From Tree-Structured Long Short-Term Memory Networks”



$$\widetilde{h} = \sum_{k \in C(t)} h_k,$$

$$i = \sigma(W_i x_t + U_i \widetilde{h} + b_i),$$

$$u = \tanh(W_u x_t + U_u \widetilde{h} + b_u),$$

$$o = \sigma(W_o x_t + U_o \widetilde{h} + b_o),$$

$$f_k = \sigma(W_f x_t + U_f h_k + b_f) \forall k \in C(t),$$

$$c_t = (i \odot u) \oplus \sum_{k \in C(t)} f_k \odot c_k,$$

$$h_t = o \odot \tanh c_t,$$

PSIMiner – пример обогащения

```
fun resolveType(node: PsiElement): String {
    return when {
        ElementType.OPERATION_BIT_SET.contains(node.elementType) -> OPERATOR
        node.parent is PsiLiteralExpression -> extractFromLiteral(node)
        node is PsiKeyword -> KEYWORD
        node is PsiIdentifier -> {
            val resolvedType = extractFromIdentifier(node)
            if (resolvedType == NO_TYPE || !splitTypes) return resolvedType
            else splitTypeToSubtypes(resolvedType).joinToString("|")
        }
        else -> NO_TYPE
    }
}

private fun extractFromLiteral(node: PsiElement): String =
    node.parentOfType<PsiLiteralExpression>()?.type?.presentableText ?: NO_TYPE
```