УДК 004.4

1.8. Использование методов интерпретации и компиляции для повышения эффективности программного обеспечения

Егунов В.А., Шабаловский В.А.
Волгоградский государственный технический университет, г. Волгоград

Статья подробно рассматривает применение методов компиляции и интерпретации кода в разработке программного обеспечения, акцентируя внимание на использовании абстрактных синтаксических деревьев (AST) для оптимизации и профилирования кода на примерах языков C++ и Python. Разъясняется процесс создания интерпретирующих профилировщиков на базе AST, которые интегрируют анализ производительности в процесс исполнения программы. Описываются этапы разработки интерпретаторов и применение инструментов, таких как Clang и Python, для работы с AST. В статье также представлены конкретные примеры построения и использования AST, демонстрирующие важность этих методов для улучшения общей эффективности программных решений.

Введение

В процессе разработки программного обеспечения (ПО) значительное внимание уделяется методам компиляции и интерпретации кода, а также применению оптимизаций и профилированию. Эти процессы играют ключевую роль в обеспечении эффективности и высокой производительности приложений [Maruthamuthu, 2023], [Ampomah, 2017].

Компиляция и интерпретация программ позволяют разработчикам преобразовывать исходный код в исполняемую форму, подходящую для конкретных архитектур и исполняющих сред. Компиляция переводит исходный код в машинный или другой формат, который может быть непосредственно исполнен процессором, в то время как интерпретация выполняет код на лету, без необходимости предварительной компиляции.

Вместе эти процессы обеспечивают не только эффективность и высокую производительность ПО, но и адаптируемость под различные платформы, что является важным аспектом в разработке современного ПО.

Создание интерпретирующего профилировщика для языка C представляет собой сложную и многогранную задачу, которая включает в себя разработку механизмов, используемых для анализа и выполнения ПО, а также сбора и анализа данных о его работе [Аветисян, 2011]. Основной целью такого профилировщика является предоставление подробной информации о производительности программ, что помогает выявлять и устранять узкие места, оптимизировать код и улучшать общую эффективность. Использование абстрактных синтаксических деревьев (AST) в создании профилировщика позволяет глубоко интегрировать анализ производительности в процесс исполнения, предоставляя уникальные возможности для анализа и оптимизации ПО.

Компиляция

Компиляторы принимают на вход целую программу и за несколько шагов преобразуют ее в исполняемый двоичный код [Rajwal, 2023], [Singla, 2014]. Полученный двоичный код можно запустить только на той системе, для которой он был скомпилирован, поскольку код привязан к конкретному оборудованию и не поддается переносу. Компиляция программы производится один раз, затем можно запускать двоичный код неограниченное количество раз. Так как компиляторы обрабатывают программы целиком, они могут обнаружить некоторые ошибки, такие как ошибки типа, синтаксиса, предупреждая о необходимости их исправления [Abelson, 1996]. Классическим примером компилируемого языка является язык C. Представим псевдокод и покажем (рис. 1), как происходит его компиляция.

```
v = 5;
x = 3;
g = v * x;
print (g);
```

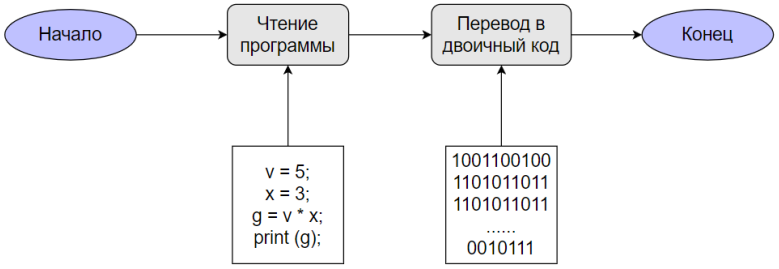

Рисунок 1 – Реализация компиляции кода

Приведенные (рис. 1) двоичные инструкции представляют собой команды для вычислительной системы. Например, присваивание «v=5;» реализуется примерно следующим набором инструкций: константа записывается в один из регистров МП, далее содержимое регистра сохраняется в память по адресу, который связывается компилятором с переменной v. Похожим образом формируются наборы

инструкций для реализации остальных операторов программы. Полученные инструкции записываются в двоичный файл, который затем может быть исполнен.

Интерпретация

Интерпретаторы работают, читая и исполняя программу по одной инструкции за раз. Каждая инструкция преобразуется в двоичный код машины и затем запускается [Introduction to Interpreters], [Федотова, 2016]. В отличие от компиляторов, интерпретаторы не создают двоичный исполняемый файл. Каждый раз при запуске программы требуется вызов интерпретатора, который затем пошагово считывает и исполняет программу. Именно поэтому интерпретатор должен присутствовать в оперативной памяти компьютера каждый раз при запуске программы, в отличие от компиляторов, которые нужны только во время компиляции. Кроме того, в отличие от компиляторов, интерпретаторы обнаруживают все ошибки во время выполнения. Примером интерпретируемого языка является Python [Abdulhamit, 2020]. Приведены этапы работы (рис. 2).

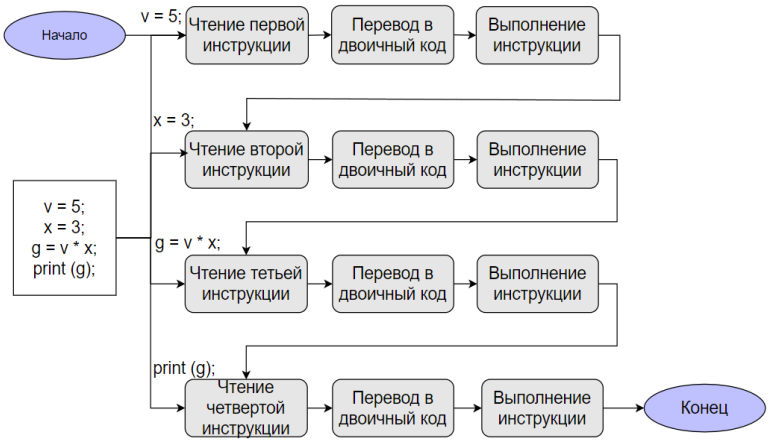

Рисунок 2 – Реализация интерпретатора

Таблица 1 – Различия компиляторов и интерпретаторов

Компилятор	Интерпретатор
Обрабатывает всю программу для получения исполняемого файла	Обрабатывает и исполняет по одной инструкции за раз
Преобразует программы в двоичный машинный код	Выполняет программы, загружая и транслируя инструкции одну за другой
Используется только один раз на этапе компиляции	Запускается при каждом выполнении программы
Позволяет обнаружить некоторые ошибки перед выполнением	Все ошибки перехватываются во время выполнения
Не нужен во время выполнения	Используется для исполнения программы
Скомпилированные программы выполняются быстрее	Интерпретируемые программы работают медленнее

Абстрактные синтаксические деревья в профилировании

AST являются мощным инструментом для представления структуры исходного кода программы. В AST каждый узел дерева соответствует определенной конструкции языка, будь то выражения, операторы, объявления переменных и функций [Пласковицкий, 2014]. Это позволяет профилировщику анализировать и модифицировать программу на более высоком уровне абстракции, что облегчает выявления сложных взаимосвязей и зависимостей в коде. Анализаторы в компиляторах играют ключевую роль в процессе анализа, трансформации и, в некоторых случаях, непосредственном исполнении кода.

Чтобы понять, как выполнить программу на основе её текста, необходимо пройти через несколько этапов, общих для компиляторов и интерпретаторов [Сидорова, 2019].

1. Первый шаг – лексический анализ и токенизация. На данном шаге текст программы преобразуется в последовательность токенов, мелких синтаксических единиц, таких как ключевые слова, идентификаторы, литералы, операторы.
2. На следующем этапе производится синтаксический анализ, где последовательность токенов преобразуется в AST. AST отображает структуру кода, включая вложенность операций, объявления переменных, определения функций и так далее. Структура AST напрямую соответствует синтаксису языка программирования.
3. Следующий шаг – семантический анализ. На данном шаге проверяются типы данных, области видимости и другие аспекты, которые не могут быть выявлены на предыдущих этапах. Например, компилятор или интерпретатор проверит, что функции вызываются с правильным числом и типами аргументов и ряд других характеристик кода.

Используемые готовые библиотеки

Существуют инструменты, которые используются для решения различных задач, связанных с анализом и рефакторингом кода, статистическим анализом безопасности или даже создания новых продуктов. Для работы с AST в контексте языка C можно выделить следующие инструменты.

1. Clang известен своими возможностями по генерации диагностических сообщений и предоставляет обширный API для работы с AST [Assembling a Complete Toolchain]. Библиотека LibClang предоставляет интерфейс для работы с AST, позволяя анализировать структуру исходного кода, проводить рефакторинг и многое другое.

2. GNU GCC (GNU Compiler Collection) [GCC online documentation]. GCC традиционно известен как компилятор, его внутренние структуры и представления, включая GIMPLE и RTL могут быть использованы для анализа и манипуляции с кодом на уровне, близком к AST. Он предлагает различные внутренние API для расширений, такие как GCC plugins, которые могут использоваться для работы с внутренним представлением кода.
3. CIL (C Intermediate Language) – это высокоуровневое представление для языка C, предназначенное для упрощения анализа и трансформации C-программ [CIL]. Представляет C-программы в виде упрощённого AST, который легче анализировать и модифицировать. CIL можно использовать для различных задач анализа кода, включая оптимизацию, инструментирование и верификацию программ на C.
4. Pycparser – это полный парсер для языка C, написанный на Python [Python parser]. Он способен разбирать исходный код на C и строить AST, что делает его полезным инструментом для задач анализа кода. Используется для написания статических анализаторов, автоматизации задач рефакторинга и т.д.
5. ANTLR (ANother Tool for Language Recognition) является мощным генератором парсеров, который может быть использован для создания парсеров, лексеров и трансляторов для различных языков, включая C [Parr, 2013]. Хотя ANTLR не предоставляет готовую библиотеку AST специфично для C, он позволяет создавать парсеры, которые могут строить AST для исходного кода. Это делает его полезным инструментом для создания специализированных инструментов анализа и трансформации кода.
6. GNU Bison и Flex представляют собой инструменты для генерации синтаксических и лексических анализаторов соответственно. Часто используются вместе для создания интерпретаторов и компиляторов. Bison генерирует парсер на основе грамматики, описанной в форме Бэкуса-Наура (БНФ или EBNF), и может автоматически создавать AST для анализируемого кода. Flex используется для создания лексического анализатора, который разбивает входной текст на токены, подготавливая их для синтаксического анализа Bison'ом.

Эти инструменты и библиотеки предоставляют мощные возможности для работы с кодом на уровне абстрактных синтаксических деревьев, обеспечивая разработчиков инструментами для анализа, трансформации и оптимизации исходного кода.

Реализация интерпретирующего профилировщика

Разработку интерпретатора можно разделить на 5 этапов.

1. Парсинг исходного кода. Процесс начинается с разбора исходного кода для построения AST. Данный этап требует точного анализа синтаксиса языка и учета всех его особенностей.
2. Аннотация AST. На этом этапе абстрактное синтаксическое дерево (AST) дополняется специальными метками или аннотациями, которые отмечают важные точки в коде для сбора информации во время выполнения программы. Эти метки могут указывать на начало и конец функций, циклов и других важных сегментов кода, что позволяет системе профилирования отслеживать выполнение и сбор данных в этих ключевых моментах.
3. Интерпретация аннотированного дерева. Интерпретатор исполняет код, следуя структуре AST, и собирает данные о времени выполнения и использования ресурсов в моменты, указанные аннотациями. Это позволяет получить детальную картинку исполнения программы.
4. Анализ данных профилирования. Собранные данные анализируются для выявления узких мест, анализа времени выполнения различных частей кода и оптимизации использования ресурсов. Результаты могут быть представлены в виде отчетов или визуализированы для удобства.
5. Оптимизация кода. На основе данных профилирования можно вносить изменения в код программы для устранения узких мест и повышения общей производительности программы.

Создание интерпретатора с использованием AST имеет несколько важных преимуществ. Возможность точно анализировать программу на уровне её логической структуры позволяет точно идентифицировать критические участки кода. Имеет хорошую гибкость, можно модифицировать для вставки профилировочного кода без изменения исходного текста программы, что обеспечивает высокую гибкость при анализе и оптимизации. Также наличие структурированного представления кода упрощает процесс сборки данных и их последующего анализа, способствуя более эффективной оптимизации.

Построение AST с использованием компилятора clang на C++ и языка программирования Python

Построение AST (абстрактного синтаксического дерева) с помощью компилятора clang на C++ и языка программирования Python начинается на этапах лексического и синтаксического анализа. В этот момент внешний интерфейс компилятора не только создаёт таблицу символов, но и проверяет корректность структуры AST. Полученный код в промежуточном представлении затем передаётся в серверную часть компилятора для дальнейшей обработки.

Переходя к более детальному изучению AST в контексте clang, важно упомянуть три основных класса, которые формируют структуру этого дерева. Это классы Decl, обозначающие объявления в коде, Stmt, отвечающие за операторы, и Type, представляющие типы данных. Эти классы являются фундаментом для создания различных подклассов и не имеют общего базового класса, что подчёркивает их независимость и специфическую роль в составе AST.

Класс ASTContext играет ключевую роль в управлении и хранении информации о всей структуре AST, содержит таблицу идентификаторов, менеджер исходного кода, список объявленных типов. Он имеет несколько связующих методов для поиска родителей узла (getParents) и обеспечивает глобальный доступ к специальным конструкциям языковых опций (getLangOpts), таким как узел стандартного size_t типа. В качестве примера используется следующая программа.

```
int main() {    vector<int> vec1 = { 1, 2, 3 };
    vector<int> vec2 = { 4, 5, 6 };
    vector<int> result = addVectors(vec1, vec2);
    cout << "Result of addition: ";
    for (int num : result) {
        cout << num << " ";
    }
    cout << endl;

    return 0;
}
```

На основе данного кода, с помощью инструмента clang производится компиляция и на выходе получается сгенерированное AST.

```
| | |-FunctionTemplateDecl 0x18bc62198e0 <line:416:5, line:417:57> col:32 isnan
| | | |-TemplateTypeParmDecl 0x18bc62196a8 <line:416:15, col:21> col:21 referenced class depth 0 index 0 _Ty
| | | `-FunctionDecl 0x18bc6219838 <line:417:20, col:57> col:32 isnan 'bool (_Ty) throw()' inline
| | |   |-ParmVarDecl 0x18bc6219758 <col:43, col:47> col:47 _X '_Ty'
| | |   `-<<<NULL>>>
| | |-FunctionTemplateDecl 0x18bc6219bf0 <line:422:5, line:423:60> col:32 isnormal
| | | |-TemplateTypeParmDecl 0x18bc62199b8 <line:422:15, col:21> col:21 referenced class depth 0 index 0 _Ty
| | | `-FunctionDecl 0x18bc6219b48 <line:423:20, col:60> col:32 isnormal 'bool (_Ty) throw()' inline
| | |   |-ParmVarDecl 0x18bc6219a68 <col:46, col:50> col:50 _X '_Ty'
| | |   `-<<<NULL>>>
| | |-FunctionTemplateDecl 0x18bc621a050 <line:428:5, line:429:76> col:32 isgreater
| | | |-TemplateTypeParmDecl 0x18bc6219cc8 <line:428:15, col:21> col:21 referenced class depth 0 index 0 _Ty1
| | | |-TemplateTypeParmDecl 0x18bc6219d50 <col:27, col:33> col:33 referenced class depth 0 index 1 _Ty2
| | | `-FunctionDecl 0x18bc6219fa8 <line:429:20, col:76> col:32 isgreater 'bool (_Ty1, _Ty2) throw()' inline
| | |   |-ParmVarDecl 0x18bc6219e00 <col:47, col:52> col:52 _X '_Ty1'
| | |   |-ParmVarDecl 0x18bc6219e80 <col:61, col:66> col:66 _Y '_Ty2'
| | |   `-<<<NULL>>>
| | |-FunctionTemplateDecl 0x18bc621a470 <line:434:5, line:435:81> col:32 isgreaterequal
| | | |-TemplateTypeParmDecl 0x18bc621a130 <line:434:15, col:21> col:21 referenced class depth 0 index 0 _Ty1
| | | |-TemplateTypeParmDecl 0x18bc621a1b0 <col:27, col:33> col:33 referenced class depth 0 index 1 _Ty2
| | | `-FunctionDecl 0x18bc621a3c8 <line:435:20, col:81> col:32 isgreaterequal 'bool (_Ty1, _Ty2) throw()' inline
| | |   |-ParmVarDecl 0x18bc621a260 <col:52, col:57> col:57 _X '_Ty1'
| | |   |-ParmVarDecl 0x18bc621a2e0 <col:66, col:71> col:71 _Y '_Ty2'
| | |   `-<<<NULL>>>
| | |-FunctionTemplateDecl 0x18bc6212170 <line:440:5, line:441:73> col:32 isless
| | | |-TemplateTypeParmDecl 0x18bc6211e30 <line:440:15, col:21> col:21 referenced class depth 0 index 0 _Ty1
| | | |-TemplateTypeParmDecl 0x18bc6211eb0 <col:27, col:33> col:33 referenced class depth 0 index 1 _Ty2
| | | `-FunctionDecl 0x18bc62120c8 <line:441:20, col:73> col:32 isless 'bool (_Ty1, _Ty2) throw()' inline
| | |   |-ParmVarDecl 0x18bc6211f60 <col:44, col:49> col:49 _X '_Ty1'
| | |   |-ParmVarDecl 0x18bc6211fe0 <col:58, col:63> col:63 _Y '_Ty2'
| | |   `-<<<NULL>>>
```

Рисунок 3 – Фрагмент сгенерированного AST

Представленный дамп показывает, как clang отображает шаблонные функции в AST. Каждая функция определяется через шаблон (FunctionTemplateDecl) с одним или несколькими типами-параметрами (TemplateTypeParmDecl), причем для каждого шаблона предоставляется одна специализация функции (FunctionDecl). В этих специализациях используется параметры-типы, заданные в шаблоне для определения функций. Данная структура позволяет C++ компиляторам, таким как clang, генерировать конкретные экземпляры функций из шаблонов в зависимости от контекста использования, обеспечивая мощные возможности для обобщенного программирования.

Для дальнейшего анализа построение AST производилось на языке Python с использованием модуля “ast”, который является стандартной библиотекой. Данный модуль помогает Python приложениям обрабатывать AST. Реализация кода программы представлена ниже.

```
def add_vectors(v1, v2):
    size = min(len(v1), len(v2))
    result = [v1[i] + v2[i] for i in range(size)]
    return result

def main():
    vec1 = [1, 2, 3]
    vec2 = [4, 5, 6]
    result = add_vectors(vec1, vec2)

    print("Result of addition: ", end="")
```

```
    for num in result:
        print(num, end=" ")
    print()

if __name__ == "__main__":
    main()
```

На (рис. 4) показан фрагмент построенного абстрактного синтаксического дерева.

```
Module(
    body=[
        FunctionDef(
            lineno=2,
            col_offset=0,
            end_lineno=5,
            end_col_offset=17,
            name='add_vectors',
            args=arguments(
                posonlyargs=[],
                args=[
                    arg(lineno=2, col_offset=16, end_lineno=2, end_col_offset=18, arg='v1', annotation=None, type_comment=None),
                    arg(lineno=2, col_offset=20, end_lineno=2, end_col_offset=22, arg='v2', annotation=None, type_comment=None),
                ],
                vararg=None,
                kwonlyargs=[],
                kw_defaults=[],
                kwarg=None,
                defaults=[],
            ),
            body=[
                Assign(
                    lineno=3,
                    col_offset=4,
                    end_lineno=3,
                    end_col_offset=32,
                    targets=[Name(lineno=3, col_offset=4, end_lineno=3, end_col_offset=8, id='size', ctx=Store())],
                    value=Call(
                        lineno=3,
                        col_offset=11,
                        end_lineno=3,
                        end_col_offset=32,
                        func=Name(lineno=3, col_offset=11, end_lineno=3, end_col_offset=14, id='min', ctx=Load()),
                        args=[
                            Call(
                                lineno=3,
                                col_offset=15,
                                end_lineno=3,
                                end_col_offset=22,
                                func=Name(lineno=3, col_offset=15, end_lineno=3, end_col_offset=18, id='len', ctx=Load()),
```

Рисунок 4 – Фрагмент AST на Python

Этот фрагмент иллюстрирует, как анализируется код и представляется в виде структурированного дерева, которое затем может быть использовано для различных целей, таких как оптимизация кода, статистический анализ, автоматическая генерация и т.д.

Для того, чтобы визуализировать дерево, требуется написать скрипт, который анализирует код, строит его AST, и затем рекурсивно обходит абстрактное синтаксическое дерево, создавая узлы и связи для визуализации с помощью Graphviz. Для этого была добавлена следующая функция.

```
def visualize_ast(node, graph=None, parent=None, name=None):
    if graph is None:
        graph = Digraph()
        graph.node_attr.update(shape='box')
```

Результатом работы является граф, иллюстрирующий структуру дерева, который может сохраняться в файл или отображаться напрямую. На рисунке 5 показана часть большого дерева.

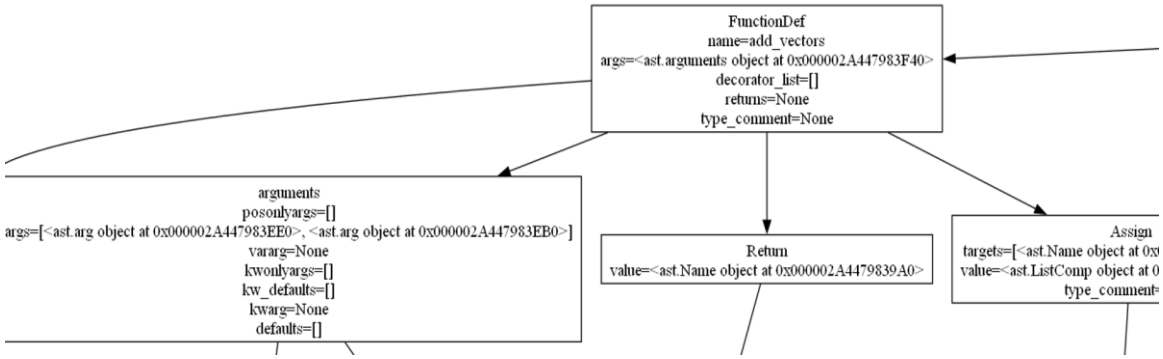

Рисунок 5 – Фрагмент визуализированного AST

В представленном фрагменте «FunctionDef» является узлом, представляющим определение функции «add_vectors», которая принимает два аргумента «v1» и «v2». Arguments представляет собой дочерний узел «FunctionDef», который содержит информацию об аргументах функции. Узлы «Asssign» предоставляют операции присваивания. Узлы «Return» отображают инструкцию возврата из функции. В данном случае функция «add_vectors» возвращает переменную «result». Эти описания соответствует следующему фрагменту кода.

```
def add_vectors(v1, v2):
    size = min(len(v1), len(v2))
    result = [v1[i] + v2[i] for i in range(size)]
    return result
```

Заключение

AST являются фундаментальным инструментом в области разработки компиляторов и профилировщиков, предоставляя детальное представление структуры исходного кода. Их применение в интерпретирующих профилировщиках, несмотря на сложность реализации, открывает широкие возможности для улучшения производительности и качества ПО. Анализ AST позволяет оптимизировать код, устраняя избыточные операции, и обеспечивает точную диагностику критических участков, что особенно ценно при работе с такими языками программирования, как C.

Эффективность профилирования заключается в способности детально настраивать производительность через анализ взаимодействий между процедурами и применение эффективных оптимизаций, например, встраивание функций и модернизацию циклов. Кроме того, автоматизация рефакторинга с помощью инструментов, основанных на AST, способствует совершенствованию практик программирования, повышению читаемости кода и, как следствие, облегчению его обслуживания. В итоге применение AST в профилировщиках и инструментах статического анализа является стратегически важным подходом для разработки высокоэффективных программных решений.

Литература

1. Аветисян А.И., Курмангалеев К.Ю., Курмангалеев Ш.Ф. (2011) Динамическое профилирование программы для системы LLVM / // Труды ИСП РАН, 2011. №21. С. 71-82.
2. Пласковицкий В.А., Урбанович П.П. (2014) Использование абстрактных синтаксических деревьев для обфускации кода // Труды БГТУ. Серия 3: Физико-математические науки и информатика, 2014. №6 (170). С. 142-146.
3. Сидорова Е.В., Дмитриева Н.Г., Калинина Н.А. (2019) Решение задачи построения графа зависимостей программных модулей в системе node.JS // Труды НГТУ им. Р. Е. Алексеева, 2019. №4 (127). С. 44-52.
4. Федотова Е.Л., Федотов А.А., Мадумарова К.В. (2016) Методика унификации процессов интерпретации программного кода // Вестник евразийской науки, 2016. №3(34). С. 142.
5. Abdulhamit S. (2020(Practical Machine Learning for Data Analysis Using Python // Academic Press, 2020, p. 534.
6. Abelson H., Sussman G.J. (1996) Structure and Interpretation of Computer Programs, second edition // MIT Press, 1996, pp.574.
7. Ampomah E. K., Mensah E., Gilbert A. (2017) Qualitative Assessment of Compiled, Interpreted and Hybrid Programming Languages // Communications on Applied Electronics. 2017 № 7, pp. 8-13.
8. Maruthamuthu R., Dhabliya D., Abbas A. H., Barno A. (2023) Advancements in Compiler Design and Optimization Techniques // E3S Web of Conferences, 2023. Vol. 399, p/4047.
9. Parr T. (2013) The Definitive ANTLR 4 Reference // Pragmatic Bookshelf 2013, pp.305.
10. Rajwal S. Chakraborty P. (2023) Application of artificial intelligence in compiler design // Proceedings of Forty-first National Conference on Recent Trends in 5G Technology & Artificial Intelligence. 2023, pp.234-239.
11. Singla T., Vashishtha V., Singh S. (2014) Compiler Construction // International Journal of Research, 2014. Vol.1, №9 URL: https://journals.pen2print.org/index.php/ijr/article/view/610.

Дополнительные источники

1. Assembling a Complete Toolchain // clang. [Электронный ресурс]. Режим доступа: https://clang.llvm.org/docs/Toolchain.html.
2. CIL - Infrastructure for C Program Analysis and Transformation (v. 1.3.7) // people.eecs.berkeley. [Электронный ресурс]. Режим доступа: https://people.eecs.berkeley.edu/~necula/cil/.
3. GCC online documentation // gcc.gnu. [Электронный ресурс]. Режим доступа: https://gcc.gnu.org/onlinedocs/gcc-13.2.0/gcc/.
4. Introduction to Interpreters // geeksforgeeks. [Электронный ресурс]. Режим доступа: https://www.geeksforgeeks.org/introduction-to-interpreters/.
5. Python parser for C, C++ // doanminhdang. [Электронный ресурс]. Режим доступа: https://doanminhdang.gitlab.io/notes/Tools/Software/Python/Python_parser_for_C,_C++.html

References in Cyrillics

1. Avetisyan A.I., Kurmangaleev K.Yu., Kurmangaleev Sh.F. Dinamicheskoe profilirovanie programmy` dlya sistemy` LLVM / // Trudy` ISP RAN, 2011. №21. S. 71-82.
2. Plaskoviczkij V.A., Urbanovich P.P. Ispol`zovanie abstraktny`x sintaksicheskix derev`ev dlya obfuskacii koda // Trudy` BGTU. Seriya 3: Fiziko-matematicheskie nauki i informatika, 2014. №6 (170). C. 142-146.
3. Sidorova E.V., Dmitrieva N.G., Kalinina N.A. Reshenie zadachi postroeniya grafa zavisimostej programmny`x modulej v sisteme node.JS // Trudy` NGTU im. R. E. Alekseeva, 2019. №4 (127). C. 44-52.
4. Fedotova E.L., Fedotov A.A., Madumarova K.V. Metodika unifikacii processov interpretacii programmnogo koda // Vestnik evrazijskoj nauki, 2016. №3(34). C. 142.

Ключевые слова:

Оптимизация и профилирование, абстрактные синтаксические деревья (AST), инструменты для работы с AST, анализ и выполнение кода, методы компиляции и интерпретации.

Егунов Виталий Алексеевич – доцент кафедры «ЭВМ и системы» Волгоградского государственного технического университета, Волгоград, Россия,
ORCID: 0000-0001-9087-3275, vegunov@mail.ru
Шабаловский Владимир Андреевич – магистрант кафедры «ЭВМ и системы» Волгоградского государственного технического университета, Волгоград, Россия,
shabalovsky.v@mail.ru

Vitaly A. Egunov, Vladimir A. Shabalovsky. Using interpretation and compilation techniques to improve software efficiency

Keywords:

Optimization and profiling, abstract syntax trees (AST), tools for working with AST, code analysis and execution, compilation and equivalence methods

DOI: 10.34706/DE-2024-03-08
JEL classification: L86 –Информация и интернет-сервисы, компьютерное программное обеспечение

Abstract

The article provides a detailed examination of the application of compilation and code interpretation methods in software development, focusing on the use of abstract syntax trees (AST) for code optimization and profiling with examples in C++ and Python. The process of creating interpreting profilers based on AST, which integrate performance analysis into the program execution process, is explained. The stages of interpreter development and the use of tools such as Clang and Python for working with AST are described. The article also presents specific examples of constructing and using AST, demonstrating the importance of these methods for improving the overall efficiency of software solutions.