УДК 004.624

П. В. Ставицький, В. В. Войтко, О. Н. Романюк

АНАЛІЗ ІНСТРУМЕНТІВ МЕТАПРОГРАМУВАННЯ В МОВАХ ПРОГРАМУВАННЯ ЗАГАЛЬНОГО ПРИЗНАЧЕННЯ

Вінницький національний технічний університет, Вінниця

Анотація. Розглянуто та проаналізовано сучасні методи метапрограмування, що використовуються у мовах загального призначення. Підхід метапрограмування використовується для багатьох сценаріїв роботи з кодом програм, важливим серед них є генерування вихідного коду. Кожна мова програмування включає окремий набір інструментарію для вирішення завдань метапрограмування. Одним з методів метапрограмування є побудова процесорів анотацій, проте тут не визначаються особливості генерування коду. Іншим методом є побудова плагінів компіляторів, що є можливим у мовах програмування типу Kotlin. Інтерфейс плагінів може надавати доступ до багатьох стадій компіляції програм, проте їх недоліком є підвищена складність програм, що значно впливає на швидкість розробки й на швидкодію роботи результуючого програмного забезпечення. Технології типу KotlinPoet та JavaPoet дозволяють генерувати текст вихідних програм з частковим дотриманням безпечної типізації на рівні інструкцій. Недоліком таких технологій є значна відмінність коду програми, що виконує генерування, від вихідного згенерованого коду, що підвищує складність роботи з цим інструментом та загальне когнітивне навантаження. На противагу їм можна використовувати рядки з механізмом інтерполяції, що забезпечують декларативність, проте тут відсутня валідація типів. Мови програмування типу MetaOCaml, Scala реалізують механізми метапрограмування й багатоетапного програмування, зокрема, на рівні синтаксису з використанням конструкцій цитування і зрощування та механізмів вбудовування і макросів. Макроси присутні також у мовах C та C++, вони за допомогою директив препроцесора дозволяють виконувати попереднє перетворення коду перед основною стадією компіляції. Коректне поєднання елементів метапрограмування дозволить створити універсальний підхід до використання функціоналу декларативного метапрограмування, що надасть потужний інструментарій для масштабування обсягів генерованого коду і підвищить якість кінцевого програмного продукту.

Ключові слова: метапрограмування, генерування коду, директиви препроцесора, макроси, Java, Kotlin, C, C++, MetaOCaml, Scala, цитування, зрощування.

Abstract. Modern methods of metaprogramming used in general-purpose languages are considered and analyzed. The metaprogramming approach is used for many scenarios of working with program code, but one of them is source code generation. Each programming language includes a unique set of tools for solving metaprogramming tasks. One of the methods of metaprogramming is the construction of annotation processors, however, they do not define the specifics of code generation. Another method is writing compiler plugins, which is possible in programming languages like Kotlin. The plugin interface can provide access to many stages of program compilation, but their disadvantage is the increased complexity of programs, which significantly affects the speed of development and the performance of the resulting software. Technologies such as KotlinPoet and JavaPoet allow generating the text of source programs with partial type safety at the level of statements. The disadvantage of such technologies is the significant difference between the code of the program that performs the generation and the original generated code, which increases the complexity of working with this tool and the overall cognitive load. In contrast to them, it is possible to use strings with an interpolation mechanism that provides declarativeness, but at the cost of type validation. Programming languages such as MetaOCaml and Scala implement mechanisms of metaprogramming and multi-stage programming, in particular, at the syntax level using quoting and splicing constructions and mechanisms of inlining and macros. Macros are also present in the C and C++ languages, they allow preprocessor directives to perform pre-transformation of the code before the main stage of compilation. The correct combination of metaprogramming elements will allow creating a universal approach to the use of a declarative metaprogramming functionality, which will provide a powerful toolkit for scaling the amount of generated code and increase the quality of the final software product.

Key words: metaprogramming, code generation, preprocessor directives, macros, Java, Kotlin, C, C++, MetaOCaml, Scala, quoting, splicing.

DOI: https://doi.org/10.31649/1999-9941-2022-55-3-44-50.

Вступ

Основною задачею метапрограмування є отримання програмного коду як вихідних даних у результаті виконання інших програм [1]. Інструменти, які реалізують підходи метапрограмування, є розповсюдженими у великій кількості мов програмування загального призначення [2]. Кожна мова реалізує певний набір функцій, а у багатьох випадках перелік інструментів метапрограмування є унікальним для окремих мов [3].

Актуальність

Використання підходів метапрограмування є розповсюдженою практикою під час розробки програмного забезпечення при використанні будь-яких мов програмування загального призначення [1-2].

Генерування коду з використанням метапрограмування дозволяє автоматизувати процес написання повторюваних фрагментів, зменшуючи кількість коду, написаного вручну, сприяє уникненню помилок та проведенню оптимізацій [3].

Переваги такого підходу є наочними під час роботи з програмним забезпеченням, що складається з мільйонів рядків коду, адже кількість необхідного згенерованого коду за допомогою різних технологій значно зростає. Тому тема статті, що присвячена аналізу інструментів метапрограмування в мовах програмування загального призначення, є актуальною з огляду на формування рекомендацій щодо ефективного використання інструментарію метапрограмування в процесі розробки масштабних програмних проєктів.

© П. В. Ставицький, В. В. Войтко, О. Н. Романюк, 2022

Мета

Підвищення ефективності розробки програмного забезпечення шляхом формулювання й використання універсального підходу до реалізації функціоналу метапрограмування, а саме генерування коду як складової мов програмування загального призначення. Універсальність полягає у тому, що такий підхід може бути застосований до будь-якої мови програмування з мінімальними синтаксичними відмінностями і дозволить досягти поставленої мети під час генерування коду.

Задачі

1. Аналіз підходів метапрограмування, що використовуються в індустрії розробки програмного забезпечення, як складових мов програмування загального призначення.
2. Розробка підходів до універсального декларативного метапрограмування, які можуть бути застосованими як інструментарій розширення до існуючих мов програмування загального призначення без необхідності модифікації їх компіляторів та інтерпретаторів.

Розв'язання задач

Більшість мов програмування загального призначення реалізують ті чи інші підходи метапрограмування. Важливо розглянути та проаналізувати їх переваги й обмеження для розробки універсального рішення щодо генерування коду, що дозволить розширити синтаксис існуючих мов програмування загального призначення.

У рамках дослідження розглянемо низку інструментів метапрограмування, що застосовуються в різних мовах програмування та реалізуються за допомогою різних методів і підходів. Важливо розглянути кожний з таких підходів та визначити компоненти, які було б доцільно використати під час генерування вихідного коду як результату виконання цільової програми (рис. 1).

Рисунок 1 − Класифікація базових інструментів метапрограмування

У процесі аналізу інструментів метапрограмування мови Java окремо виділимо рефлексію. Рефлексія − це функція мови Java, що дозволяє програмам досліджувати або аналізувати себе та маніпулювати внутрішніми властивостями програми. Наприклад, клас Java може отримати імена всіх своїх методів і відобразити їх.

Метод рефлексії дозволяє виконувати певні модифікації програмного коду під час його виконання, але рефлексія не надає можливості генерування тексту нового коду для виконання компілятором пізніше. Крім того, використання рефлексії сповільнює процес виконання програми, тому не завжди є бажаним рішенням для поставлених завдань у метапрограмуванні.

Мови Java та Kotlin можуть використовувати анотації також для генерування тексту програмного коду. Процесори анотацій типу APT, KAPT та KSP дозволяють додавати логіку генерування коду, виходячи з відповідних компонентів програмних засобів, що вже використовують наперед визначені анотації [1]. Приклад програми, що виконує генерування декларованої сигнатури функції на основі відповідного інтерфейсу, наведено на рисунку 2.

На рисунку 2 зображено два фрагменти коду, де перший є складовою програми, що виконує генерування коду, а другий демонструє результуючий згенерований код. Декларативність такого підходу полягає в тому, що можна прослідкувати певну схожість між двома наведеними фрагментами коду. Перший фрагмент описує сигнатуру згенерованої функції. Список полів інтерфейсу з першого фрагменту є ідентичним списку аргументів згенерованої функції. Проте такий підхід є лише частково-декларативним, оскільки не дозволяє з аналогічним успіхом генерувати тіла функцій, вирази, здійснювати динамічне декларування функцій з використанням аргументів для варіативної кількості аргументів згенерованої функції й динамічного імені функції.

```
@Function(name = "example_function")
interface ExampleFunction {
    val arg1: String
    val arg2: Map<String, List<*>>
}

>
fun myAmazingFunction(
    arg1: String,
    arg2: Map<String, List<*>>
) {
    // ...
}
```

Рисунок 2 − Приклад використання процесора анотацій для частково декларативного генерування коду мови Kotlin

З іншого боку, метод декларативного метапрограмування може бути застосований під час написання процесора анотацій. При застосуванні декларативного підходу метапрограмування генерування коду відбуватиметься з використанням відповідних шаблонів, що описують загальну структуру генерованого коду, дозволяючи модифікувати певні деталі для кожної ітерації (рис. 3).

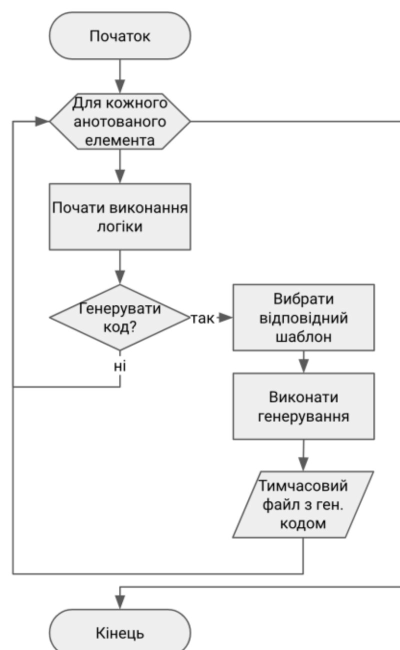

Рисунок 3 − Алгоритм генерування коду в ході роботи процесора анотацій з використанням декларативного методу метапрограмування на основі шаблонів

На рисунку 3 продемонстровано місце декларативного метапрограмування в процесі написання процесорів анотацій. Ключовим елементом є застосування декларативних шаблонів, що можуть бути використані кожною з ітерацій обробки певного анотованого елемента. Шаблони можуть містити код, що є ідентичним до бажаного згенерованого, і дозволяти використовувати аргументи для деталізації окремих складових результуючого коду. Використання процесорів анотацій відповідає на питання "Коли генерувати код?", проте як саме відбуватиметься цей процес — залишається на розсуд розробника.

Технології JavaPoet та KotlinPoet у поєднанні з процесорами анотацій можуть підвищити ефективність генерування коду. Їх інтерфейс дозволяє досягти чіткої типізації на стадії написання логіки генерування коду, що дозволяє попередити значну кількість помилок на ранніх стадіях розробки. Приклад генерування тривіальної програми за допомогою KotlinPoet наведено на рисунку 4.

На рисунку 4 наведено приклад генерування простої програми, яка друкує повідомлення в консоль. Верхня частина рисунку демонструє фрагмент програми, що виконує генерування, а нижня частина містить згенерований код. Як можна побачити, код між такими фрагментами суттєво відрізняється навіть при написанні простої програми. Недоліком технології KotlinPoet є те, що фрагменти коду, які містять логіку генерування, значно відрізняються від вихідного коду, що підвищує когнітивне навантаження під час роботи з кодом та ускладнює його підтримку. Тому пропонуємо при використанні декларативного

підходу застосовувати шаблони коду зі збереженням високого рівня валідації типів, аналогічно до KotlinPoet.

```
val file = FileSpec.builder("", "Example")
  .addFunction(
    FunSpec.builder("main")
      .addStatement("println(%P)", "Example print!")
      .build()
  )
  .build()
file.writeTo(System.out)

>
fun main() {
    println("Example print!")
}
```

Рисунок 4 − Приклад імперативного генерування коду з KotlinPoet, що демонструє суттєву різницю між кодом програми, що генерує код, та вихідним кодом

Розглянувши проблему з іншого боку, зазначимо, що використання технології ktlint [2] може слугувати іншою точкою входу для генерування коду. На відміну від процесора анотацій, що виконує свою роботу під час компілювання коду, ktlint є статичним аналізатором коду, завданням якого є знаходження помилок у стилі написання програм та забезпечення автоматизації рефакторингу коду. В основі роботи з таким інструментом лежить набір правил, що валідує коректність використання тих чи інших конструкцій мови програмування Kotlin. Виконання рефакторингу відбувається окремо від стадії компіляції та виконання програми, що дозволяє ефективно генерувати код у тих випадках, коли початковий варіант не задовольняє стилістичним вимогам. Тоді проводиться аналіз абстрактного синтаксичного дерева (АСД), що будується на основі існуючого коду. Маючи доступ до АСД, існує можливість генерування нового коду з модифікаціями. Приклад створення власного правила для таких задач наведено на рисунку 5.

```
class CodeModificationRule : Rule("code-modification") {
  override fun visit(
    node: ASTNode,
    autoCorrect: Boolean,
    emit: (offset: Int, errorMessage: String, canBeAutoCorrected: Boolean) -> Unit
  ) {
        // Генерування коду на основі компонентів синтаксичного дерева
    }
  }
}
```

Рисунок 5 − Приклад побудови правила ktlint для модифікування/генерування коду

Рисунок 5 демонструє приклад використання технології ktlint під час генерування коду та створення власного набору правил для цього. Існує можливість створення власного правила для вирішення завдань з генерування коду. Для цілей модифікації коду можна використовувати стандартний інтерфейс ktlint, що забезпечує доступ до абстрактного синтаксичного дерева існуючого коду в функціях типу visitor, що надає можливість його модифікації. Як і процесор анотацій, ktlint надає можливість визначити, коли саме необхідно виконувати генерування, проте він не надає інструментів для підвищення ефективності такого процесу. Декларативний підхід метапрограмування може бути застосовано як один з інструментів для генерування коду за допомогою ktlint. Під час створення декларативних шаблонів коду аргументами для них можуть бути компоненти синтаксичного дерева коду перед модифікацією (ASTNode). Аналогічні інструменти ktlint доступні на інших платформах у вигляді feross/standard для JavaScript, gofmt для мови програмування Go.

Компілятори мов програмування також виконують генерування коду в процесі компілювання для оптимізації програм. Синтаксис багатьох мов програмування забезпечує відповідні елементи для виконання такого генерування коду. Наприклад, мови програмування типу Kotlin, Scala, C++ використовують модифікатор inline [3] як вбудовування. Це усуває накладні витрати на з'єднання викликів і відкриває значні можливості для оптимізації. Оскільки функції з модифікатором inline не будуть використані за

посиланням, а їх тіло буде буквально скопійоване у місця їх виклику, схожий механізм можна використати для підходу декларативного метапрограмування. Скопійований фрагмент коду є коректним компільованим кодом, що, в свою чергу, надає усі функції статичної типізації та валідації. Якщо застосовувати такий код як шаблон і використати його як основу для генерування нового коду, можна забезпечити декларативність зі збереженням коректної типізації. Потрібно забезпечити варіативність шаблонів залежно від переданих аргументів та умов для кожної з ітерацій генерування нових файлів або фрагментів коду. Отримати доступ до копіювання таких шаблонів можна за допомогою власного плагіна компілятора, що може дозволити аналізувати і модифікувати внутрішню структуру коду програми в процесі її компіляції.

Мови програмування C та C++ використовують аналогічний підхід перед етапом роботи компілятора, а не під час його роботи. Таке рішення може бути досягнуто за допомогою макросів та директив препроцесора [4]. Функціонал макросів можна також розглянути в якості механізму для генерування коду на основі шаблонів зі схожими можливостями до зрощування. Проте замість доступу до внутрішньої структури коду тут можна виконати попередню його обробку та модифікацію. Таким чином, існує можливість додавання власного синтаксису до будь-якої з існуючих мов програмування. За допомогою власного препроцесора можна модифікувати код, який будує шаблони, у такий, що буде зрозумілим оригінальному компілятору мови програмування.

Функціональна мова програмування OCaml початково не містить потужних засобів метапрограмування та генерування коду. Проте її розширення під назвою MetaOCaml [5] вирішує цю задачу шляхом використання концепцій цитування (quoting) та зрощування (splicing) [6]. Підхід є схожим до механізму інтерполяції рядків у мовах програмування типу Kotlin, де під час декларування семантики рядка можна динамічно вбудовувати посилання на змінні або вирази, значення яких буде отримано під час виконання програми. Аналогічно MetaOCAml дозволяє поєднувати функціональну логіку з вбудованими декларативними фрагментами генерованого коду.

Якщо використовувати підхід цитування та зрощування в якості розширення існуючих мов програмування, їх синтаксис може різнитися в залежності від особливостей конкретної мови. Проте, новий функціонал генерування коду додано як розширення оригінального компілятора без офіційних його оновлень.

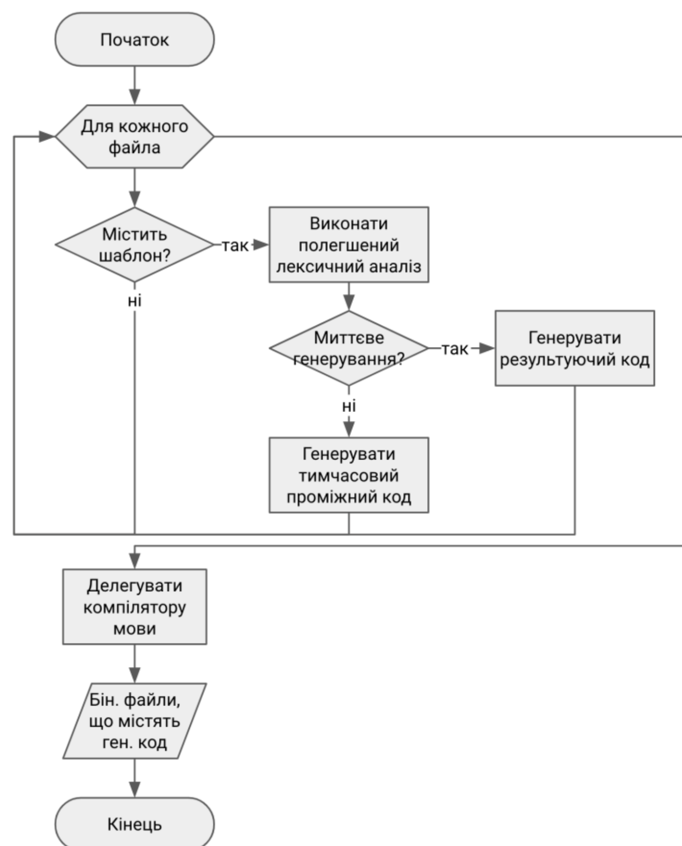

Рисунок 6 − Алгоритм використання макросів у процесі генерування коду

Цитування та зрощування зазвичай розширює синтаксис мов програмування. Під час розробки методу метапрограмування очевидно немає доступу до модифікації компіляторів існуючих мов програмування. Використовуючи підхід, аналогічний до макросів, існує можливість попередньої обробки коду, яка реалізує модифікований синтаксис і генерування проміжного коду, що є зрозумілим для оригінального компілятора мови програмування. Далі компілятор матиме змогу згенерувати бажаний код як результат виконання завдання. Такий алгоритм може використовуватися для кожного файлу, що містить розширений синтаксис цитування та зрощування, генеруючи проміжний код та делегуючи його компілювання і виконання оригінальному компілятору цільової мови програмування (див. рис. 6).

Під час обробки доданого синтаксису до існуючої мови програмування немає необхідності в повному синтаксичному і лексичному аналізі коду, тому спрощеного лексичного аналізу з визначенням символів цитування та зрощування буде достатньо. Далі існує можливість миттєвого генерування коду. У випадку, коли потрібна можливість відкладеного генерування коду під час виконання програми, генерується проміжний відповідний код у вигляді функцій, які генеруватимуть код на вимогу. Крім того, ті файли з кодом, що не містять додаткового синтаксису, можуть повністю ігноруватися.

Результати порівняльного аналізу методів метапрограмування на предмет можливості повернення згенерованого коду як вихідних даних та безпеки типів наведено у таблиці 1.

Таблиця 1 - Порівняльний аналіз методів метапрограмування

Метод метапрограмування	**Надають можливість повертати згенерований код як вихідні дані**	**Забезпечують безпеку типів**
Рефлексія	ні	так
Плагіни компілятора	ні	так
Процесори анотацій	частково	частково
Технологі типу KotlinPoet	так	так
Статичні аналізатори коду	частково	частково
Вбудовування	ні	так
Макроси	так	так
Підхід інтерполяції рядків	так	ні
Підхід цитування та зрощування	так	так

Методи, які використовують процесори анотацій та статичні аналізатори коду, ідентифіковані значенням "частково", оскільки вони забезпечують платформу/фреймворк для генерування коду, але не визначають деталей. Це означає, що для генерування коду необхідно додатково обрати інший метод метапрограмування.

Висновки

Для створення ефективного інструмента генерування коду, що вирішує проблемні моменти наведених рішень, необхідно сфокусуватися на поєднанні різних можливостей розглянутих підходів до реалізації декларативного методу метапрограмування з дотриманням безпечної типізації.

Процесори анотацій мов програмування Kotlin або Java можуть бути використані для часткової реалізації методу декларативного метапрограмування. Під час розробки таких процесорів бажано використовувати інші технології, що надають потрібний функціонал. Схожим чином статичні аналізатори коду можуть організувати фреймворк для генерування коду, але не уточнювати, як саме це необхідно робити.

Методи зрощування та макросів мають багато спільного і дозволяють виконувати генерування/копіювання коду, проте їх ключовою відмінністю є стадія компіляції, протягом якої вони виконуються. Використання плагінів компілятора може надати доступ до внутрішнього стану програми під час компілювання, що дозволить використання певних кодових конструкцій як шаблонів для генерування інших фрагментів коду. Також макроси дозволяють надати можливість використання декларативних шаблонів коду окремо від стадії компілювання, дозволяючи розширювати синтаксис розповсюджених мов програмування за допомогою компонентів цитування та зрощування.

Таким чином, коректне поєднання елементів метапрограмування дозволить створити ефективний інструмент для вирішення завдань генерування коду, надаючи універсальний підхід до його застосування для великої кількості мов програмування загального призначення.

References

[1] Juárez-Martínez, Ulises and José Oscar Olmedo-Aguirre, *Annotation Processing as Local Variable Crosscutting*. 2008.

[2] Grégory Lureau, *Ktlint: a great 1st experience*. 2020. [Online]. Available: https://www.glureau.com/2020/05/26/Ktlint-Moshi-Introduction/. Accessed on: Sept. 4, 2022.

[3] Theodoros Theodoridis, Tobias Grosser, and Zhendong Su. "Understanding and exploiting optimal function inlining," *27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '22) (2022)*. Association for Computing Machinery, New York, NY, USA, 977–989.

[4] Medeiros, Flávio M., Márcio Ribeiro, Rohit Gheyi, Sven Apel, Christian Kästner, Bruno Ferreira, Luiz Carvalho and Baldoino Fonseca, "Discipline Matters: Refactoring of Preprocessor Directives in the #ifdef Hell," *IEEE Transactions on Software Engineering*, 44 (2018), pp. 453-469.

[5] Oleg Kiselyov, "The Design and Implementation of BER MetaOCaml − System Description," *FLOPS (2014).*

[6] Pouya Larjani, "On Meta Programming and Code Generation in F#," 2010/4/2.

Стаття надійшла: 20.09.2022.

Відомості про авторів

Ставицький Павло Валерійович − аспірант кафедри програмного забезпечення.

Войтко Вікторія Володимирівна − кандидат технічних наук, доцент кафедри програмного забезпечення.

Романюк Олександр Никифорович − доктор технічних наук, професор, завідувач кафедри програмного забезпечення.

P. V. Stavytskyi, V. V. Voitko, O. N. Romanyuk

ANALYSIS OF METAPROGRAMMING CAPABILITIES IN GENERAL-PURPOSE PROGRAMMING LANGUAGES

Vinnytsia National Technical University, Vinnytsia