DOI 10.31891/2307-5732-2022-311-4-249-255
УДК 004.624

СТАВИЦЬКИЙ П. В.
Вінницький національний технічний університет
https://orcid.org/0000-0002-9139-6076
e-mail: pavlo.stavytskyi@gmail.com
ВОЙТКО В. В.
Вінницький національний технічний університет
https://orcid.org/0000-0002-3329-7256
e-mail: defakfki@i.ua

МЕТОД ДЕКЛАРАТИВНОГО МЕТАПРОГРАМУВАННЯ НА ОСНОВІ ПРЕДМЕТНО-ОРІЄНТОВАНИХ МОВ ПРОГРАМУВАННЯ

Розглянуто концепцію декларативного метапрограмування, а саме генерування програмного коду, що базується на використанні предметно-орієнтованих мов. Декларативність полягає у тому, що вхідний код, який відповідає за генерування коду за синтаксичними ознаками, є якомога ближчим до вихідного результуючого коду. В якості демонстрації методу розглянуто програмну систему для генерування коду мови програмування Starlark, що є діалектом Python. У рамках дослідження концепція декларативного метапрограмування реалізується на базі предметно-орієнтованої мови, що імітує синтаксичні конструкції мови Starlark. Розглянуто компоненти мови Starlark, реалізовані за допомогою предметно-орієнтованої мови на базі Kotlin, зокрема, інструкції присвоєння, спискові вирази, вирази конкатенації, оператор нарізання масивів, виклики функцій тощо.

Ключові слова: декларативне програмування, метапрограмування, генерування коду, предметно-орієнтована мова, Kotlin, Starlark, Python.

Pavlo STAVYTSKYI, Viktoriia VOITKO
Vinnytsia National Technical University

METHOD OF THE DECLARATIVE METAPROGRAMMING BASED ON DOMAIN-SPECIFIC PROGRAMMING LANGUAGES

A concept of declarative metaprogramming which focuses on program code generation is considered. The method is based on using domain-specific languages that are implemented by means of existing general-purpose programming languages. Since metaprogramming is based on using programs that produce other programs, the declarative approach means that the input and the output program code must be as close as possible in terms of syntax or ideally identical. In order to generate a program code, it is required to write templates that hold the exact structure of a resulting generated code but contain placeholders for specifics of concrete generated instances. Such an approach is highly useful when there is a need of generating a large number of similar program files by structure but different in details. The simplest way to define such templates would be just by constructing them from string literals by injecting the specific details using a string interpolation mechanism for example. However, such an approach does not provide type safety and compile time validation, so there is no guarantee the generated code is a correct and compilable program.

The domain-specific language, on the other hand, replicates all the syntactic constructions of the target programming language so that, the resulting generated code would be the same as code written by the programmer. Moreover, having preserved all the syntax validations by the compiler on which the domain-specific language is based. The paper considers a concrete example of a code generator for a Starlark programming language, a dialect of Python while the domain specific-language is based on Kotlin language. Using the toolset of Kotlin, such a domain-specific language replicates such Starlark constructs as variable assignments, list comprehensions, library functions, slices, etc with a required level of customizability.

Keywords: declarative programming, metaprogramming, code generation, domain-specific language, Kotlin, Starlark, Python.

Постановка проблеми у загальному вигляді та її зв'язок із важливими науковими чи практичними завданнями

Сучасна індустрія розробки програмного забезпечення часто опирається на генерування програмного коду для підвищення надійності програмних продуктів та пришвидшення їх розробки. Під час генерування коду для систем різного призначення необхідно використовувати потужний та надійний інструмент, що зменшить вірогідність допущення помилок на ранніх стадіях. Таким чином, інструмент повинен забезпечувати валідацію типів та синтаксичну схожість вхідного і вихідного коду.

Аналіз останніх джерел

Дослідження [1] фокусується на розробці системи Piranha, спрямованої на автоматичний рефакторинг застарілих гілок кодової бази, що пов'язані з використанням вмикачів функціоналу (feature flag). Така система використовує рішення google java format для реформатування java коду під стилі коду, що підтримуються компанією Google. Для мови Swift, наприклад, система використовує власні реформатори коду, що здебільшого фокусуються на операторах потоку керування. Недоліком складових генерування коду в такому рішенні є те, що, зазвичай, код програми, який описує генерування, значно відрізняється від результуючого згенерованого коду й ускладнює процес розробки та підтримки програм.

Cyclone [2] – це мова програмування, що базується на синтаксисі C та реалізує функціонал генерування коду на базі шаблонів. Тут пропонуються додаткові синтаксичні конструкції для побудови шаблонів. Недоліком є те, що функціонал генерування коду доступний лише для мови Cyclone, що унеможливлює використання такого рішення для робочих сценаріїв, де використовуються розповсюджені мови програмування.

Формулювання цілей статті

Метою роботи є удосконалення підходів генерування коду мов програмування шляхом застосування концепції декларативного метапрограмування з використанням шаблонів. Дане дослідження фокусується на реалізації, що використовує предметно-орієнтовану мову на основі існуючої мови програмування загального призначення.

Виклад основного матеріалу

При роботі з системами генерування програмний код можна розділити на дві складові:

- вхідний код – програмний код, вихідними даними якого є інший програмний код, який необхідно згенерувати;
- вихідний код – програмний код, який згенеровано в результаті виконання вхідного коду.

Запропоновано підхід декларативного метапрограмування для існуючих мов програмування, що, на відміну від існуючих рішень, забезпечує ідентичність вхідного та вихідного коду при створенні шаблонів генерування коду зі збереженням валідації типів на стадії компіляції. Зокрема, запропонований метод фокусується на використанні предметно-орієнтованих мов як базису для реалізації декларативного метапрограмування.

Розглянемо програмну систему для генерації коду мови програмування Starlark [3], що є діалектом мови Python та використовується як мова написання скриптів для системи збірки проєктів Bazel [4]. Вхідний код такої системи написано за допомогою мови програмування загального призначення Kotlin. Завдяки особливостям синтаксису цієї мови доступний потужний інструментарій для створення предметно-орієнтованих мов програмування (ПОМ).

Дослідження фокусується на розробці ПОМ на базі синтаксичних конструкцій мови програмування Kotlin, що реалізує підхід декларативного метапрограмування для генерування коду мови Starlark.

Особливість ПОМ полягає у відтворенні синтаксичних конструкцій мови Starlark таким засобами мови Kotlin, як: перевантаження операторів; лямбда вирази; лямбда-вирази з отримувачами; делегування властивостей класів; функції, що відтворюють стандартну бібліотеку Starlark та Bazel тощо. На рисунку 1 наведено приклад такої ПОМ.

```kotlin
// Kotlin
fun file_template(
    appName: String,
    srcFiles: List<String>,
    mainClass: String

) = BUILD.bazel {
    load("@rules_java//java:defs.bzl", "java_binary")

    java_binary(
        name = appName,
        srcs = list["MyClass.java"] `+` srcFiles,
        main_class = mainClass,
    )
}
```

Рис. 1. Приклад шаблону, що використовує Starlark ПОМ на базі мови Kotlin

Генерування коду відбувається за допомогою шаблонів. Для створення шаблону необхідно декларувати звичайну функцію та обернути її тіло за допомогою BUILD.bazel функції, що відповідає за створення однойменного файлу та є вхідною точкою в контекст предметно-орієнтованої мови для генерування Starlark коду. Створена функція називається шаблоном, адже вона реалізує однойменний шаблон проєктування [5], дозволяючи декларувати загальний скелет згенерованого файлу, залишаючи місця для вставлення аргументів, притаманних конкретному згенерованому файлу. Використання шаблонів дозволяє автоматизувати випадки, де необхідна генерація великої кількості одноманітних файлів, загальний вміст яких є базовим з введенням конкретних деталей.

У результаті виконання такої програми з конкретними аргументами буде згенеровано Starlark код, як зображено на рисунку 2.

Варто зазначити, що код на рисунку 2 згенеровано у відформатованому вигляді зі збереженням необхідних відступів та пробілів. Тому згенерований код є легко читабельним та зрозумілим. Такий результат досягається шляхом генерування абстрактного синтаксичного дерева (АСД) [6] Starlark коду, що необхідно згенерувати.

```
# Starlark
load("@rules_java//java:defs.bzl", "java_binary")

java_binary(
    name = "myApp",
    srcs = ["MyClass.java"] + [
        "Class1.java",
        "Class2.java"
    ],
    main_class = "Main",
)
```

Рис. 2. Приклад згенерованого Starlark коду в результаті використання шаблону на базі ПОМ

Структура даних групує усі синтаксичні компоненти коду у деревовидну структуру, зручну для використання при форматуванні та генеруванні коду. Для наведеного прикладу під час виконання Kotlin коду, що використовує Starlark ПОМ, отримуємо абстрактне дерево, зображене на рисунку 3.

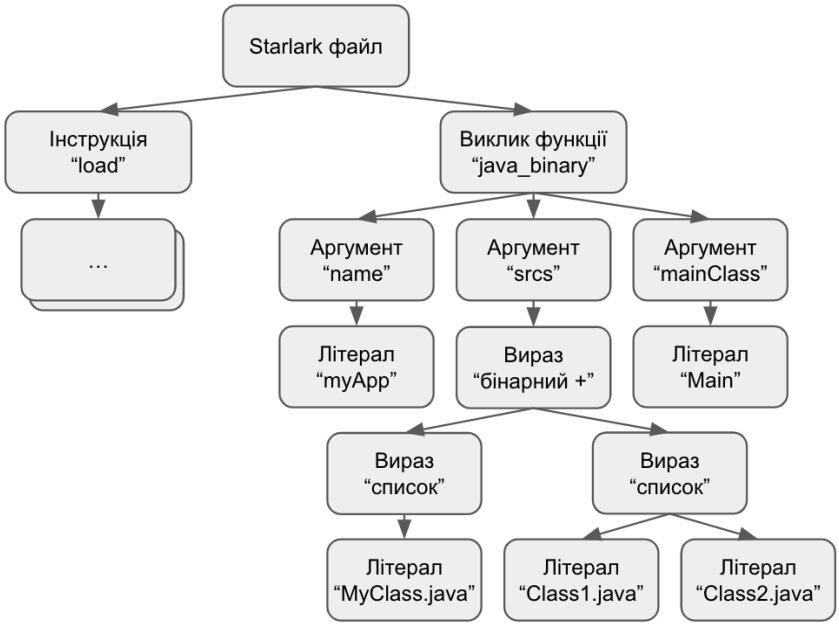

Рис. 3. Абстрактне синтаксичне дерево, побудоване в результаті виконання Kotlin коду на базі Starlark ПОМ

Результатом роботи ПОМ є абстрактне синтаксичне дерево коду мови програмування Starlark, концептуально ідентичне до оригінального АСД, що будується інтерпретаторами мови Starlark.

Існує три розповсюджені інтерпретатори мови Starlark, написані мовами Java (у складі кодової бази системи збірки проєктів Bazel), Go та Rust.

З точки зору інтерпретатора Starlark розроблений ПОМ замінює собою перші два кроки інтерпретації коду – лексичний і синтаксичний аналіз. Тому АСД, побудоване за допомогою Starlark ПОМ, може бути використане для будь-яких цілей, включаючи подальші кроки інтерпретації, такі як семантичний аналіз. Проте основною областю застосування такого рішення є генерування форматованого коду. Модель загального процесу генерації коду зображена на рисунку 4.

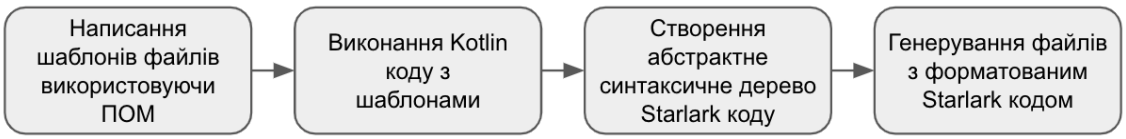

Рис. 4. Модель процесу генерування Starlark коду з використанням Starlark ПОМ на базі мови Kotlin

Синтаксичні можливості ПОМ мають обмеження, які визначаються реалізацією за допомогою синтаксичних конструкцій мови Kotlin. Тому ПОМ з огляду на синтаксис не є повністю ідентичною з мовою Starlark. Проте мова Kotlin серед усіх інших мов програмування загального призначення пропонує найбільшу ступінь гнучкості, що дозволяє досягни якомога ближчої схожості ПОМ та Starlark.

Далі розглянемо синтаксичні конструкції мови програмування Starlark, що відтворені у ПОМ на базі мови Kotlin.

Для реалізації інструкції присвоєння значення змінній у ПОМ використано оператор by мови Kotlin,

а саме його перевантаження, що дозволяє реалізувати власний функціонал. На рисунку 5 зображено приклад такої операції.

```
// Kotlin
val LIST by list["item1", "item2"]
```

```
// Згенерований Starlark код
LIST = ["item1", "item2"]
```

Рис. 5. Реалізація інструкції присвоєння в ПОМ

Варто зазначити, що інструкція присвоєння зберігає типізацію в ПОМ. Таким чином, якщо далі така змінна використовуватиметься в ПОМ, компілятор Kotlin провалідує коректність ПОМ відповідно до правил типізації мови Starlark. Такий підхід дозволяє запобігти низці помилок у вихідному коді на ранніх етапах роботи. Серед основних типів, які підтримуються, виділимо: рядки, числа, булеві значення, словники та списки.

Іншим важливим компонентом мови Starlark є спискові вирази (list comprehensions) [7], що дозволяють зручно створювати колекції елементів. Приклад використання такого виразу в ПОМ наведено на рисунку 6.

```
// Kotlin
val LIST_1 by list["item1", "item2"]
val LIST_2 by ("i" `in` LIST_1 take { i -> i })
```

```
// Згенерований Starlark код
LIST_1 = ["item1", "item2"]
LIST_2 = [i for i in LIST_1]
```

Рис. 6. Реалізація спискових виразів у ПОМ

Важливим у такій конструкції є збереження виразом типізації. *LIST_1* є змінною типу “список рядків”, а тому при інтегруванні змінна *i* на рівні Kotlin та ПОМ матиме тип “рядок”. Такий підхід забезпечує чіткий рівень валідації з боку компілятора мови Kotlin.

Часто важливо не лише передати дані у вигляді змінної або значення, а й скомпонувати значення, користуючись кількома джерелами даних, наприклад, побудувати список шляхом складання декількох інших списків. Для таких випадків мова Starlark реалізує вирази конкатенації, що необхідно також відтворити в розроблюваній ПОМ. Для цього необхідно реалізувати функцію з ім’ям `+`.

```
// Kotlin
val LIST_1 by list["item1", "item2"]
val LIST_2 by ("i" `in` LIST_1 take { it `+` "_modified" }) `+` LIST_1

// Приклад помилки компіляції.
// Очікується List<StringType>, але отримано List<Any>.
val LIST_3: List<StringType> by LIST_2 `+` list[1, 2, 3]
```

```
// Згенерований Starlark код
LIST_1 = ["item1", "item2"]
LIST_2 = [i + "_modified" for i in LIST_1] + LIST_1
```

Рис. 7. Реалізація виразів конкатенації в ПОМ

Варто зазначити, що в такому випадку в ПОМ використовується не оператор “плюс”, а звичайна функція з таким же ім’ям, індикатором чого є зворотні галочки, в які огорнута назва функції. Використання звичайного оператора “плюс” у ПОМ було б не достатньо, оскільки така операція виконувалася б на рівні Kotlin, а вихідний Starlark результат містив би вже результуючий список. Проте необхідною є можливість саме генерування виразу конкатенації на рівні вихідного коду, для чого використовується функція `+`. На рисунку 7 продемонстровано приклад реалізації такого виразу.

З рисунку 7 можна побачити, що вираз конкатенації зберігає типізованість, дозволяючи валідувати коректність вхідного коду на рівні Kotlin.

Аналогічно оператор нарізання масивів (array slising) мови Starlark реалізується в ПОМ за допомогою оператора діапазону (range operator) з мови Kotlin. Приклад нарізання масивів наведено на рисунку 8.

```
// Kotlin
"abc.kt"[0..-3]

// Згенерований Starlark код
"abc.kt"[0:-3]
```

Рис. 8. Реалізація оператора нарізання масивів у ПОМ

Основною синтаксичною одиницею в мові Starlark та системі Bazel є інструкція виклику функцій або декларування таргетів (build target), що в результаті дозволяє компілювати та виконувати відповідний програмний код за допомогою Bazel.

ПОМ містить стандартну бібліотеку розповсюджених функцій в Starlark та Bazel. Декларація їх виклику виконується звичайним викликом таких функцій, що продемонстровано на рисунку 9.

```
// Kotlin
android_binary(
    name = "app",
    srcs = list[...],
    manifest_values = dict {
        "minSdkVersion" to "21"
        "targetSdkVersion" to "29"
    }
)
```

Рис. 9. Приклад виклику функцій в ПОМ

Функції, що використовуються у Starlark та Bazel, можуть бути оновлені розробниками в процесі їх життєвого циклу. Існує можливість опинитися в ситуації, коли функції ПОМ не міститимуть усіх необхідних аргументів для використання. Для забезпечення гнучкості в таких випадках ПОМ реалізує процес частково динамічного виклику функцій. Такий підхід полягає в тому, що під час генерування коду користувач має змогу додавати будь-які нові аргументи в існуючі функції. Для цього необхідно викликати функцію з допомогою фігурних дужок замість круглих, декларувати назву необхідного аргумента за допомогою звичайного рядка та використати функцію-оператор “присвоєння” з ПОМ, оточену зворотними галочками `=`, як продемонстровано на рисунку 10.

```
// Kotlin
android_binary {
    name = "app"
    srcs = list[...]
    "manifest_values" `=` {
        "minSdkVersion" to "21"
        "targetSdkVersion" to "29"
    }
}
```

Рис. 10. Приклад частково динамічного виклику функцій в ПОМ з власними додатковими аргументами

Аналогічно до попереднього прикладу ПОМ надає можливість декларування викликів власних функцій, що не присутні в стандартній бібліотеці, як наведено на рисунку 11.

У цьому випадку сценарій схожий до попереднього. Назва функції, яку необхідно викликати, декларується в якості рядка. У разі використання в ній фігурних дужок викликається перевантажений оператор виклику функцій в Kotlin, що працює як розширення до типу String та є декларованим на вищому рівні. Сигнатура такої функції продемонстрована на рисунку 12.

```kotlin
// Kotlin
"android_binary" {
    "name" `=` "app"
    "srcs" `=` list[...]
    "manifest_values" `=` {
        "minSdkVersion" to "21"
        "targetSdkVersion" to "29"
    }
}
```

Рис. 11. Приклад динамічного виклику додаткових власних функцій у ПОМ

```kotlin
operator fun String.invoke(body: FunctionCallContext.() -> Unit)
```

Рис. 12. Сигнатура Kotlin функції, що декларує динамічний виклик Starlark функцій у ПОМ

Усі розглянуті компоненти ПОМ можуть бути поєднані для генерування більш складних прикладів коду Starlark, як наведено на рисунку 13.

```kotlin
// Kotlin
val KOTLIN_SOURCE_FILES by list [...]

"file" `in` KOTLIN_SOURCE_FILES take { file ->
    genrule(
        name = "modify_" `+` file[0..-3],
        srcs = list[file],
        outs = list[file[0..-3] `+` "_modified.kt"],
        cmd = "..."
    )
}
```

```starlark
// Згенерований Starlark код
KOTLIN_SOURCE_FILES = [...]

[
    genrule(
        name = "modify" + file[0:-3],
        srcs = [file],
        outs = [file[0:-3] + "_modified.kt"],
        cmd = "...",
    )
    for file in KOTLIN_SOURCE_FILES
]
```

Рис. 13. Приклад динамічного виклику додаткових власних функцій у ПОМ

Запропонований метод декларативного метапрограмування набув практичного застосування в розробленій системі Airin для автоматичного мігрування програмних продуктів на систему збірки проєктів Bazel [8].

Висновки з даного дослідження і перспективи подальших розвідок у даному напрямі

Предметно-орієнтовані мови, створені на базі мов програмування загального призначення, можуть стати інструментом для реалізації концепції декларативного метапрограмування, дозволяючи досягти якомога більшої схожості вхідного та вихідного коду, зберігаючи валідацію типів. Саме таким прикладом є ПОМ на базі мови Kotlin, що дозволяє імітувати синтаксичні конструкції мови Starlark для генерування відповідного вихідного коду.

Література

1. Ramanathan M., K., Clapp, L., Barik, R., Sridharan M. (2020). Piranha: Reducing Feature Flag Debt at Uber. https://manu.sridharan.net/files/ICSE20-SEIP-Piranha.pdf

2. SMITH, F., GROSSMAN, D., MORRISETT, G., HORNOF, L., & JIM, T. (2003). Compiling for template-based run-time code generation. Journal of Functional Programming, 13(3), 677-708. doi:10.1017/S095679680200463X

3. Starlark Language. https://bazel.build/rules/language

4. Bazel. https://bazel.build/

5. Gamma, E., Helm, R. and Johnson, R., 1994. Design Patterns: Elements of Reusable Object-Oriented Software. Addison Wesley.

6. Aho, A. V. (2007). Compilers: Principles, techniques, and tools. Pearson Addison-Wesley.

7. The Python Language Reference. https://docs.python.org/3/reference/

8. Stavytskyi P. (2021). Automated migration from Gradle to Bazel with Airin. BazelCon 2021, Google. https://youtu.be/dz-CFEwJuko

References

1. Ramanathan M., K., Clapp, L., Barik, R., Sridharan M. (2020). Piranha: Reducing Feature Flag Debt at Uber. https://manu.sridharan.net/files/ICSE20-SEIP-Piranha.pdf

2. SMITH, F., GROSSMAN, D., MORRISETT, G., HORNOF, L., & JIM, T. (2003). Compiling for template-based run-time code generation. Journal of Functional Programming, 13(3), 677-708. doi:10.1017/S095679680200463X

3. Starlark Language. https://bazel.build/rules/language

4. Bazel. https://bazel.build/

5. Gamma, E., Helm, R. and Johnson, R., 1994. Design Patterns: Elements of Reusable Object-Oriented Software. Addison Wesley.

6. Aho, A. V. (2007). Compilers: Principles, techniques, and tools. Pearson Addison-Wesley.

7. The Python Language Reference. https://docs.python.org/3/reference/

8. Stavytskyi P. (2021). Automated migration from Gradle to Bazel with Airin. BazelCon 2021, Google. https://youtu.be/dz-CFEwJuko

Рецензія/Peer review : 12.06.2022 р. Надрукована/Printed :02.08.2022 р.