DOI: https://doi.org/10.36910/6775-2524-0560-2022-47-11
УДК 004.4`4
Іваненко Антон Романович, аспірант,
https://orcid.org/0000-0002-9846-537X
Марченко Олександр Іванович, к.т.н., доцент,
https://orcid.org/0000-0002-4537-3420
Національний технічний університет України «Київський політехнічний інститут імені Ігоря Сікорського», м. Київ, Україна

СПОСІБ КОМПІЛЯЦІЇ ЗАМИКАННЯ ВКЛАДЕНИХ ФУНКЦІЙ МОВИ TYPESCRIPT У ПРОМІЖНУ МОВУ CIL ПЛАТФОРМИ .NET

Іваненко А.Р., Марченко О.І. Спосіб компіляції замикання вкладених функцій мови TypeScript у проміжну мову CIL платформи .NET. У даній статті запропонований спосіб, який дозволяє ефективно компілювати замикання вкладених функцій з захопленими змінними у проміжну мову CIL платформи .NET. Розглянутий спосіб базується на ідеї зміни проміжного представлення програми шляхом створення класу, який зберігає значення замкнених змінних, і передається останнім аргументом у вкладену функцію. Це дозволяє зв'язати виклик функції з її захопленими змінними. Генерація CIL інструкцій відбувається вже для перетвореної програми. Результатом роботи способу є згенерований код, який демонструє кращу швидкодію, ніж код, що генерується існуючим аналогом.
Ключові слова: транслятор, проміжне представлення програми, вкладена функція, замикання, CIL, CLR, .NET, TypeScript.

Ivanenko A.R., Marchenko O.I. Compilation the closure of TypeScript nested function into Common Intermediate Language of .NET platform. This article proposes a method that allows you efficiently compile the closure of nested functions with captured variables into the Common Intermediate Language of the .NET platform. The considered method is based on the idea of transforming the intermediate representation of the program by creating a structure that stores the values of closed variables and passes them to the last argument in the nested function. This allows you to associate a function call with its captured variables. CIL instructions are generated for the converted program. The result of the method is the generated code, which shows better performance than the code generated by the existing analog.
Keywords: translator, intermediate representation of the program, nested function, closure, CIL, CLR, .NET, TypeScript.

Постановка наукової проблеми.

Мова програмування JavaScript та типізована її версія TypeScript дозволяють створювати програми не тільки в парадигмі об'єктно-орієнтовно програмування, а і у функціональній також. Однією з головних особливостей функціонального програмування є можливість створення замикання за допомогою вкладених функцій. Завдяки наявності замикання, наприклад, можливо реалізувати каррінг функцій, що користується значною популярністю під час розробки програмного забезпечення на цих мовах програмування.

Варто зазначити, що дані мови є інтерпретуючими, тобто такими, трансляція та виконання програм яких відбувається інтерпретатором. Реалізація замикання при такому підході не викликає значних складнощів. Але зовсім інша річ реалізувати замикання під час трансляції у мову нижчого рівня не інтерпретатором, а компілятором, що є значно важчим.

Отже, при розробці компілятора мови програмування TypeScript у проміжну мову CIL віртуальної машини .NET постає задача розробити ефективний спосіб компіляції замикання вкладених функцій у відповідні інструкції проміжної мови CIL.

Аналіз досліджень.

На жаль, досить незначна кількість статей розкриває способи трансляції різних мов програмування у проміжну мову CIL платформи .NET. Серед них можна виділити опис методів трансляції байт-коду віртуальної машини Java [1] від корейських дослідників та трансляції мови програмування Scheme [2] від французьких вчених, які було розглянуто і проаналізовано у попередній статті [3], яка описує спосіб трансляції конкатенації рядкових виразів мови TypeScript.

Стаття [2] наводить такі три ідеї, на основі яких можна виконати трансляцію замикання вкладених функції:

1) використання делегатів;
2) використання додаткового класу;
3) використання вказівників на функції.

Варто наголосити, що дана стаття не розкриває детальний спосіб, як саме можливо реалізувати компіляцію замикання з використанням додаткового класу для збереження захоплених змінних, що дозволяє детально дослідити цей спосіб і викласти отримані результати у цій статті.

Метою даної статті є розробка способу компіляції замикання вкладених функцій мови програмування TypeScript у проміжну мову CIL платформи .NET та порівняльний аналіз швидкодії згенерованого за розробленим способом коду з результатом роботи обраного аналогу попередніх досліджень [3-4] – компілятором JScript.

Термінологія.

Вкладена функція – функція, яка визначена в контексті іншої функції.

Замикання – це доступ з вкладеної функції до змінних зовнішньої функції, у якій вона оголошена.

Захоплена змінна – змінна зовнішньої функції, яка використовується у вкладеній функції.

Проміжне представлення програми – дерево розбору, яке не містить інформацію про синтаксис та токени, а містить лише необхідну частину для виконання генерації коду.

Тип-значення – це тип, змінна (екземпляр) якого містить значення цього типу, яке зберігається у стеку [5]. Наприклад, це типи int та bool.

Тип-посилання – це тип, змінна (екземпляр) якого містить у якості свого значення посилання (вказівник) на змінну (екземпляр) цього типу, що вже містить значення цього типу, яке зберігається у кучі. Саме ж посилання (тобто адреса змінної) зберігається у стеку [5]. Наприклад, це тип string.

Ловерінг – модифікація проміжного представлення програми з метою реалізації одних конструкцій мови програмування через використання інших.

Каррінг – метод обчислення функції від багатьох аргументів перетворенням її в послідовність функцій одного аргумента[7].

Запропонований спосіб трансляції.

Розглянемо вхідну програму, записану мовою програмування TypeScript, яка містить замикання і яку потрібно транслювати в CIL (рисунок 1):

```
function main(): number {
    let a = 0;
    function test(t: number): void {
        a = t;
    }
    test(1);
    return 0;
}
```

Рисунок 1 – Вхідна програма на мові TypeScript, яка містить замикання.

Основною проблемою є те, що мова програмування CIL не підтримує вкладені функції, таким чином функція test повинна бути переміщена на вищий рівень з новим іменем. Але ускладнює таке перетворення наявність замикання (змінна a, яка використовується в контексті вкладеної функції). Для того щоб виконати таке перетворення, необхідно передати цю змінну у вкладену функцію під час виклику, але таким чином, щоб функція могла змінювати значення змінної і в контексті зовнішньої функції. Для цього необхідно, щоб незалежно від того, чи є змінна типом-значенням, чи типом-посиланням, її значення зберігалось у кучі, а внутрішня функція мала можливість змінити це значення за переданим посиланням. Якщо значення змінної буде зберігатись у стеці, то під час передачі його у функцію, віртуальна машина платформи .NET це значення скопіює, що зробить не можливим змінити контекст зовнішньої функції у внутрішній функції.

Таким чином, необхідну поведінку можливо отримати шляхом введення проміжного класу, який буде зберігати всі замкнені змінні контексту зовнішньої функції. Екземпляр цього класу буде зберігатись у кучі і передаватись до вкладеної функції за посиланням. Такий підхід дозволить вкладеній функції змінювати зовнішній контекст. Приклад такої перетвореної програми зображено на рисунку 2.

```
class $main_context {
    a: number;
}

function main(): number {
    const $context = new $main_context();
    $context.a = 0;
    $main_test(1, $context);
    alert($context.a) // a = 1
    return 0;
}

function $main_test(t: number, $context: $main_context): void {
    $context.a = t;
}
```

Рисунок 2 – Модифікована програма з рис 1, яка може бути компільована в мову CIL.

Запропонований спосіб компіляції мови програмування TypeScript у проміжну мову CIL (зазначимо, що зараз не існує прямого компілятора мови TypeScript, а існує лише через проміжну трансляцію у JavaScript) відбувається у такі етапи: лексичний, синтаксичний та семантичний аналізи; ловерінг та оптимізації; генерація коду (рис. 3). Саме на етапі ловерінгу можливо модифікувати написану користувачем програму. Для цього необхідно виконати певні перетворення над деревом розбору, що містить проміжне представлення цієї програми.

Запропонований спосіб компіляції замикання вкладених функцій полягає в наступному:

1. Проаналізувати дерево розбору зовнішньої функції, обійшовши всі вкладені функції, та запам'ятати всі змінні зовнішньої функції, що використовуються у вкладених функціях
2. Згенерувати новий тип даних з унікальним іменем (щоб уникнути конфліктів з користувацькими типами) та оголосити в ньому змінні, як поля, що були отримані на етапі 1.
3. Модифікувати всі вкладені функції таким чином: додати параметр, який має тип, отриманий на попередньому етапі, а також замінити доступ до замкнених змінних, як доступ до відповідного поля класу.
4. Модифікувати зовнішню функцію наступним чином: оголосити на початку функції змінну, яка має тип, згенерований на етапі 2, замінити доступ до замкнених змінних, як доступ до відповідних полів класу, модифікувати виклик вкладених функцій, передавши оголошену змінну

останнім аргументом.

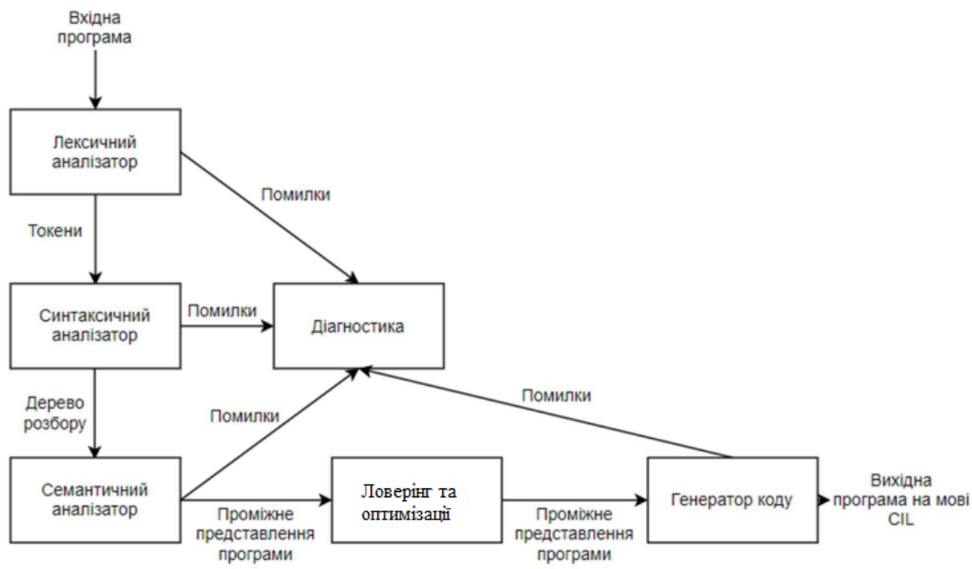

Рисунок 3 –Запропонована схема компіляції вхідної програми на мові TypeScript у проміжну мову CIL

5. Винести всі внутрішні функції на верхній рівень призначивши їм нове унікальне ім'я (щоб уникнути конфліктів з користувацькими функціями), замінити виклики цих функцій у зовнішній функції використовуючи нові імена.
6. Передати модифіковане проміжне представлення програми до генератору коду для генерації відповідних CIL інструкцій.
7. Кінець

Для генерації виконуваних файлів платформи .NET, як і у минулому дослідженні [3], було використану бібліотеку Mono.Cecil [8], яка надає зручний інтерфейс для запису інструкцій CIL. Лістинги функції, що компілюють клас та функцію у проміжну мову CIL (після виконання ловерінгу) відповідно зображено на рисунках 4, 5 і 6.

```
private void EmitClassDeclaration(ClassSymbol @class)
{
    var objectType = ResolveCLRType(TypeSymbol.Any);
    var classDef = new TypeDefinition("", @class.Name, TypeAttributes.NestedPrivate, objectType);
    foreach (var field in @class.Fields)
    {
        var typeRef = ResolveCLRType(field.Type);
        var fieldDef = new FieldDefinition(field.Name, FieldAttributes.Public, typeRef);
        classDef.Fields.Add(fieldDef);
    }

    _knownTypes.Add(@class, classDef);
    _assemblyDefinition.MainModule.Types.Add(classDef);
}
```

Рисунок 4 – Лістинг методу, що відповідає за генерацію IL інструкцій оголошення класу

Кожен метод відповідає за генерацію IL інструкцій для певної вершини дерева розбору. EmitClassDeclaration генерує інструкцій, які заповнюють мета-інформацію у збірці. Ця інформація містить типи даних, оголошені користувачем, що дозволяє використовувати їх у інших збірках, якщо вони мають відповідний модифікатор доступу (public). EmitFunctionDeclaration відповідає за генерацію оголошення функцій, як статичного методу класу, а EmitFunctionImplementation вже безпосередньо за генерацію інструкцій для тіла функції. Цей метод генерує інструкції для кожного з виразів тіла, а також слідкує за мітками для цих інструкції.

```
private void EmitFunctionDeclaration(FunctionSymbol function)
{
    var clrType = ResolveCLRType(function.Type);
    var method = new MethodDefinition(function.Name, MethodAttributes.Static | MethodAttributes.Private, clrType);

    foreach (var param in function.Parameters)
    {
        var typeRef = ResolveCLRType(param.Type);
        var paramDef = new ParameterDefinition(param.Name, ParameterAttributes.None, typeRef);
        method.Parameters.Add(paramDef);
    }

    _programTypeDefinition.Methods.Add(method);
    _methods.Add(function, method);
}
```

Рисунок 5 – Лістинг методу, що відповідає за генерацію IL інструкцій оголошення функції

```
private void EmitFunctionImplementation(FunctionSymbol function, BoundBlockStatement body)
{
    _locals.Clear();
    _fixLabels.Clear();
    _labels.Clear();

    var method = _methods[function];
    var ilProcessor = method.Body.GetILProcessor();

    foreach(var statement in body.Statements)
        EmitStatement(ilProcessor, statement);

    // replaces out internal labels to
    // IL labels
    foreach (var fix in _fixLabels)
    {
        var targetInstructionIndex = _labels[fix.Target];
        var targetInstruction = method.Body.Instructions[targetInstructionIndex];
        var instructionToFix = method.Body.Instructions[fix.InstructionIndex];
        instructionToFix.Operand = targetInstruction;
    }

    method.Body.OptimizeMacros();
}
```

Рисунок 6 – Лістинг методу, що відповідає за генерацію IL інструкцій імплементації функції

Результат роботи запропонованого способу можна побачити на рисунку 7. Наочно видно, що для вхідної програми з рисунка 1, було згенеровано відповідні інструкції для створення контексту зовнішньої функції main, що оголошують змінну відповідного класу ($main_context), та присвоюють значення замкненої змінної a, як модифікацію відповідного поля цього класу. Потім контекст передається до внутрішньої функції $main_test за посиланням.

```
IL_0002: ldc.i4.0
IL_0003: stfld int32 Program/'$main_context'::a
IL_0008: ldc.i4.1
IL_0009: ldloca.s 0
IL_000b: call void Program::'$main_test'(int32, Program/'$main_context'&)

. . . . . .

.method assembly hidebysig static
    void '$main_test' (
        int32 t,
        Program/'$main_context'& ''
    ) cil managed
{
    // Method begins at RVA 0x206e
    // Code size 8 (0x8)
    .maxstack 8

    IL_0000: ldarg.1
    IL_0001: ldarg.0
    IL_0002: stfld int32 Program/'$main_context'::a
    IL_0007: ret
}
```

Рисунок 7 –Частковий лістинг згенерованих інструкцій для прикладу, що містить замикання вкладених функцій за запропонованим способом для програми, показаної на рис.1.

Аналіз отриманих результатів.

Для тестування результатів дослідження було використано консольну програму на мові програмування C#, яка запускає виконувані файли та вимірює час їх виконання. Для більшої наочності та зменшення похибок при старті програми, виміри часу виконання коду прикладів виклику функції, яка містить замикання, генерованого різними компіляторами, проводилися в циклі на певну кількість ітерацій (рисунок 9).

Виміри проводилися для замикання вкладеністю 3 рівні та кількістю ітерацій 1000000, 10000000 та 100000000. Отримані результати показано на рисунку 9. Як видно з діаграми, код, що згенерований згідно запропонованого способу, працює значно швидше, ніж згенерований існуючим аналогом.

```
function main() {

    let a = 10;
    let b = 'test'

    function level1(t1: number)  {
        function level2(t2:number) {
            let c = b;
            function level3(t3: number) {
                let d = c;
                b = d;
            }
            level3(t1);
        }
        level2(t1);
    }

    for (let i = 0; i < 10000; i++)
        level1(i);

    return 0;
}
```

Рисунок 8 –Приклад вхідної програми на мові TypeScript, яка тестувалась.

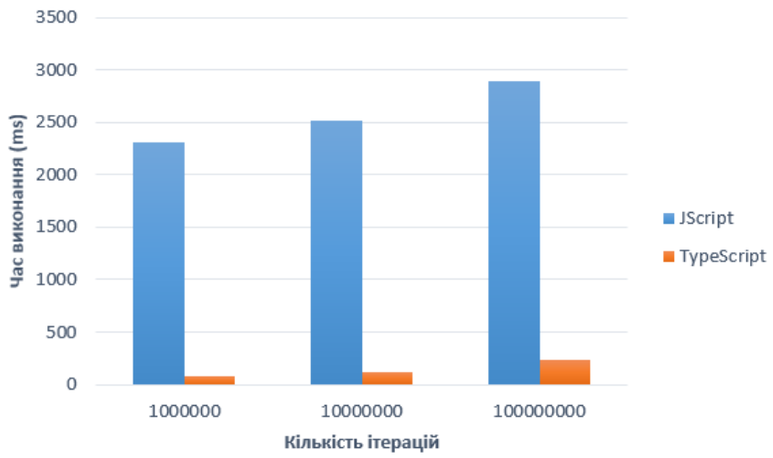

Рисунок 9 –Графік порівняння швидкодії згенерованих інструкцій для замикання компілятором JScript та компілятором TypeScript за запропонованим способом.

Висновки.

При продовженні дослідження способів компіляції мови TypeScript у проміжну мову CIL платформи .NET було розроблено спосіб компіляції замикання вкладених функцій за допомогою введення допоміжного класу для збереження контексту замикання.

В результаті реалізації описаного у цій статті способу у власному компіляторі мови TypeScript, отриманий результат під час тестування продемонстрував значно кращу швидкодію згенерованого коду в порівнянні з результатом роботи обраного аналогу – компілятором JScript.

Отже, враховуючи отримані результати, а також результати минулих досліджень [3-4], можна зазначити, що є доцільним створення прямого компілятора мови програмування TypeScript у проміжний код CIL платформи .NET.

Запропоноване дослідження має як наукову новизну, так і практичну цінність.

Список бібліографічного опису

1. YangSun Lee, SeungWon Na, DaeHoon Hwang, Language Translator for Execution Java programs in .NET [Електроний ресурс] – Режим доступу: https://www.koreascience.or.kr/article/JAKO200411922370411.pdf
2. Yannis Bres, Bernard Serpette, Manuel Serrano, Compiling Scheme programs to .NET Common Intermediate Language [Електроний ресурс] – Режим доступу: https://www.researchgate.net/publication/213885643_Compiling_Scheme_programs_to_NET_Common_Intermediate_Language.
3. Іваненко А.Р., Марченко О.І Спосіб трансляції конкатенації рядкових виразів мови TypeScript у проміжну мову CIL
4. Марченко О.І., Іваненко А.Р. Аналіз способів трансляції мови TypeScript у проміжну мову CIL платформи .NET платформи .NET. [Електроний ресурс] – Режим доступу: http://cit-journal.com.ua/index.php/cit/article/view/233/322
5. Рихтер Дж. CLR via C#. Программирование на платформе Microsoft .NET Framework 4.5 на языке C#. 4-е изд., СПб.:Питер, 2013
6. Марченко О.І. Конспект лекцій з дисципліни «Іженерія програмного забезпечення-1. Основи проектування трансляторів», Київ 2013.
7. Currying. Вікіпедія. [Електронний ресурс] – Режим доступу:https://en.wikipedia.org/wiki/Currying
8. Документація Mono.Cecil [Електронний ресурс] – Режим доступу: https://cecil.pe/

References

1. YangSun Lee, SeungWon Na, DaeHoon Hwang, Language Translator for Execution Java programs in .NET [Electronic resource] – Access mode: https://www.koreascience.or.kr/article/JAKO200411922370411.pdf
2. Yannis Bres, Bernard Serpette, Manuel Serrano, Compiling Scheme programs to .NET Common Intermediate Language [Electronic resource] – Access mode: https://www.researchgate.net/publication/213885643_Compiling_Scheme_programs_to_NET_Common_Intermediate_Language.
3. Ivanenko A.R., Marchenko O.I Translation the concatenation of TypeScript string expressions into Common Intermediate
Language of .NET platform. [Electronic resource] – Access mode: http://cit-journal.com.ua/index.php/cit/article/view/233/322
4. Marchenko O.I., Ivanenko A.R. Analysis of TypeScript translation methods into Common Intermediate Language of .NET platform
5. Jeffrey Richter. CLR via C#. 2012
6. Marchenko O.I Synopsis of lectures on the subject "Software Engineering-1. Basics of translator design », Kyiv 2013.
7. Currying. Wikipedia. [Electronic resource] – Access mode:https://en.wikipedia.org/wiki/Currying
8. Mono.Cecil documentation [Electronic resource] – Access mode: https://cecil.pe/