DOI: https://doi.org/10.36910/6775-2524-0560-2023-52-10
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

МЕТОД КОМПІЛЯЦІЇ ТИПІВ ОБ'ЄДНАННЯ МОВИ TYPESCRIPT У ПРОМІЖНУ МОВУ CIL ПЛАТФОРМИ .NET

Іваненко А.Р., Марченко О.І. Метод компіляції типів об'єднання мови TypeScript у проміжну мову CIL платформи .NET. У даній статі запропонований метод, який дозволяє ефективно компілювати змінні типу об'єднання у проміжну мову CIL платформи .NET. Розглянутий метод базується на ідеї зміни проміжного представлення програми шляхом перетворення оголошення змінної типу об'єднання у оголошення змінної узагальненого класу TSUntion, який зберігає інформацію про можливі типи змінної, а також її значення. Генерація CIL інструкцій відбувається вже для перетвореної програми враховуючи особливості введеного класу. Результатом роботи запропонованого способу є згенерований код, який показує швидкодію кращу за результат роботи компілятора мови F# у проміжну мову CIL.
Ключові слова: компілятор, проміжне представлення програми, тип об'єднання, CIL, CLR, .NET, TypeScript, JavaScript

Ivanenko A.R., Marchenko O.I. Method of compilation union types of TypeScript into Common Intermediate Language of .NET platform. This article proposes a method to efficiently compile union-type variables into the CIL intermediate language of the .NET platform. The considered method is based on the idea of changing the intermediate representation of the program by converting the declaration of the variable of the union type into the declaration of the variable of the generalized TSUntion class, which stores information about the possible types of the variable, as well as its value. The generation of CIL instructions takes place already for the converted program, taking into account the features of the introduced class. The result of the proposed method is the generated code, which shows a performance better than the result of the compiler of the F# language into the CIL intermediate language.
Keywords: compiler, intermediate representation of the program, union type, CIL, CLR, .NET, TypeScript, Javascript

Постановка наукової проблеми.

Мова програмування TypeScript дозволяє розробнику писати програми як в парадигмі об'єктно-орієнтовно програмування, так і у парадигмі функціонального. Однією з головних особливостей функціонального програмування є можливість оголошення змінних типу об'єднання. Цей тип змінних декларує, що значення відповідної змінної може мати один з перелічених типів. Завдяки цьому, наприклад, можливо реалізувати повернення значення з функції, яке може мати значення одного з декількох типів.

Програми написані на мові програмування TypeScript спочатку транслюються у мову JavaScript, яка відноситься до мов інтерпретуючого типу, тобто виконання написаних програм на цій мові відбувається інтерпретатором. Також варто зазначити, що у мові програмування JavaScript відсутня строга типізація, що значно полегшує трансляцію у цю мову програмування. На відміну від JavaScript, проміжна мова CIL віртуальної машини платформи .NET має строгу типізацію і чітко визначені правила роботи з різними типами, що значно ускладнює компіляцію мови TypeScript безпосередньо у проміжну мову CIL.

Таким чином, при розробці прямого компілятора мови програмування TypeScript у проміжну мову CIL віртуальної машини .NET постає задача розробити коректний та ефективний метод компіляції типів об'єднання у відповідні інструкції проміжної мови CIL.

Аналіз досліджень.

На сьогоднішній день існує досить незначна кількість статей, що описують способи трансляції різних мов програмування у проміжну мову CIL платформи .NET чи інші мови байт-кодового типу. Серед таких статей можна виділити опис методів трансляції байт-коду віртуальної машини Java [1] від корейських дослідників та трансляції мови програмування Scheme [2] від французьких вчених. Ці дослідження було розглянуто і проаналізовано у попередніх статтях [3,4,5], які присвячені компіляції

конкатенації рядкових виразів та замикання вкладених функцій. Відсутність великої кількості публікацій дає значний простір для проведення досліджень, які будуть мати як наукову новизну, так і практичну цінність.

Варто зазначити, що через відсутність аналогів компілятору мови TypeScript у проміжну мову CIL, які б підтримували трансляцію типів об'єднання, для порівняння отриманого результату було вирішено обрати компілятор функціональної мови F#, який підтримує схожу функціональність.

Метою даної статті є розробка методу компіляції типів об'єднання мови програмування TypeScript у проміжну мову CIL платформи .NET та порівняльний аналіз швидкодії згенерованого за розробленим способом коду з результатом роботи обраного аналога.

Термінологія.

Лексичний аналізатор – це частина транслятора, яка відповідає за перетворення вхідного тексту програми, яка представлена рядком символів, в рядок токенів (лексем), а також виконує перевірку на наявність лексичних помилок [6].

Синтаксичний аналізатор – це частина транслятора, що здійснює декомпозицію вхідної програми поданої у вигляді рядка токенів у структурні одиниці мови (оператори, описи, декларації, тощо) згідно заданої граматики і перевіряє на наявність синтаксичний помилок [6].

Семантичний аналізатор – це частина транслятора, що здійснює перевірку вхідної програми на наявність семантичних помилок, а також будує проміжне представлення програми.

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

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

Тип об'єднання – це тип, що описує значення, яке може бути одним із кількох типів. В мові Typescript використовується вертикальна риска (|), щоб розділити кожен тип. Наприклад number | string | boolean — це тип значення, яке може бути числом, рядком або логічним значенням [8].

Узагальнення (англ. generic) – засіб мов програмування, що дозволяє створювати програмний код, який містить єдине (типізоване) рішення задачі для різних типів, з його подальшим застосуванням для будь-якого конкретного типу (int, float, char тощо).

Нотація Бекуса-Наура (Форма Бекуса-Наура, БНФ) – це спосіб запису правил контекстно-вільної граматики, форма опису формальної мови.

Запропонований метод компіляції.

Через те, що початково мова програмування TypeScript була спроектована для трансляції у мову програмування JavaScript, яка не має строгої типізації, мова TypeScript має складну систему типів, яка дозволяє достатньо гнучко працювати з типізацією і покривати різні сценарії. До особливостей її системи типів можна віднести підтримку таких груп типів даних:

- узагальнені типи;
- типи об'єднання;
- типи перетину;

В даній статті розглянуто метод компіляції типів об'єднання, але зроблено це у спрощеному вигляді. Через те, що цільова мова програмування CIL платформи .NET має строгу систему типів і через те, що досить важко підтримувати усі функціональні можливості мови програмування TypeScript під час компіляції, було вирішено не підтримувати можливість виклику спільних функцій чи доступу до спільних полів обох типів. При роботі з типами об'єднання в запропонованій реалізації компілятора необхідно обов'язково привести тип об'єднання до конкретного типу за допомогою оператора instanceof, метод компіляції якого теж розглянуто у цій статті. Розглянемо метод компіляції на прикладі програми, яку зображено на рисунку 1.

```
const numberOrString: number | string = 10;
if (numberOrString instanceof string)
{
    alert('This is string: ', numberOrString);
}
else
{
    alert('This is number: ', numberOrString);
}
```

Рисунок 1 – Лістинг програми на мові TypeScript для якої виконується компіляція

Визначимо формальний опис змінних типу об'єднання мови TypeScript у формі БНФ, а також оператора instanceof та додамо підтримку цих правил до існуючого лексичного та синтаксичного аналізатора. БНФ оголошення змінної матиме такий вигляд (для простоти прикладу, вказана робота лише з примітивними типами мови TypeScript):

<variable-declarations-statement> --> <variable-declarations>
<variable-declarations> --> <declaration-keyword> <declarations-list>
<declarations-list> --> <declaration> | <declaration>,<declarations-list>
<declaration> --> <identifier> <type-annotation>
<type-annotation> --> :<type-clause>
<type-clause> --> <attribute> | <union-type>
<union-type> --> <type-clause><union-delimiter><type-clause>
<attribute> --> boolean | number | string
<declaration-keyword> --> var | let | const
<union-delimiter> --> |

БНФ підтримки оператора instanceof:

<expression> --> <instanceof-expression>
<instanceof-expression> --> <literal-expression> instanceof <type-expression>
<literal-expression> --> <literal>
<type-expression> --> <attribute>

Після проведення лексичного і синтаксичного аналізу вхідна програма на мові TypeScript буде перетворена у проміжну форму у вигляді дерева розбору. На етапі семантичного аналізу необхідно проаналізувати усі блоки коду (тобто тіла функцій, циклів та умовних конструкцій) та перевірити, що для будь-якої змінної типу об'єднання ми звертаємось до її методів та полів лише попередньо перевіривши її на ознаку, що вона має значення певного типу. Для виконання таких дій на етапі семантичного аналізу та ловерінгу пропонується наступний метод:

1. Якщо змінна в даному контексті має декілька можливих типів, то під час доступу до її полів, вивести оголошення про помилку.

2. Якщо наявна умовна конструкція if з перевіркою на тип змінної (instanceof), то в тілі умовної конструкції оголосимо додаткову змінну, яка буде мати тип на який виконувалась перевірка і присвоюємо їй значення поточної змінної, попередньо виконавши операцію приведення типу до відповідного їй типу. Всі операції з поточною змінною в межах блоку переписуємо, використовуючи нововведену змінну.

3. Якщо при виконанні пункту 2, умовна конструкція if має блок else, то у випадку, коли змінна має лише два можливих типи, потрібно виконати ті ж самі дії, як у пункті 2, але для протилежного типу. Якщо ж змінна має більше, ніж 2 типи, то виключити тип, що був перевірений для контексту змінної у області видимості цього блоку.

4. Якщо виконується перевірка на тип, який не є одним з можливих типів, що змінна може мати, то виконуємо видалення усього блоку коду, оскільки він є недосяжним при виконанні програми.

Таким чином після виконання семантичного аналізу та ловерінгу отримуємо фінальне представлення програми, яке можна транслювати у проміжну мову CIL платформи .NET. Приклад перетворення проміжного представлення програми за запропонованим способом зображено на рисунку

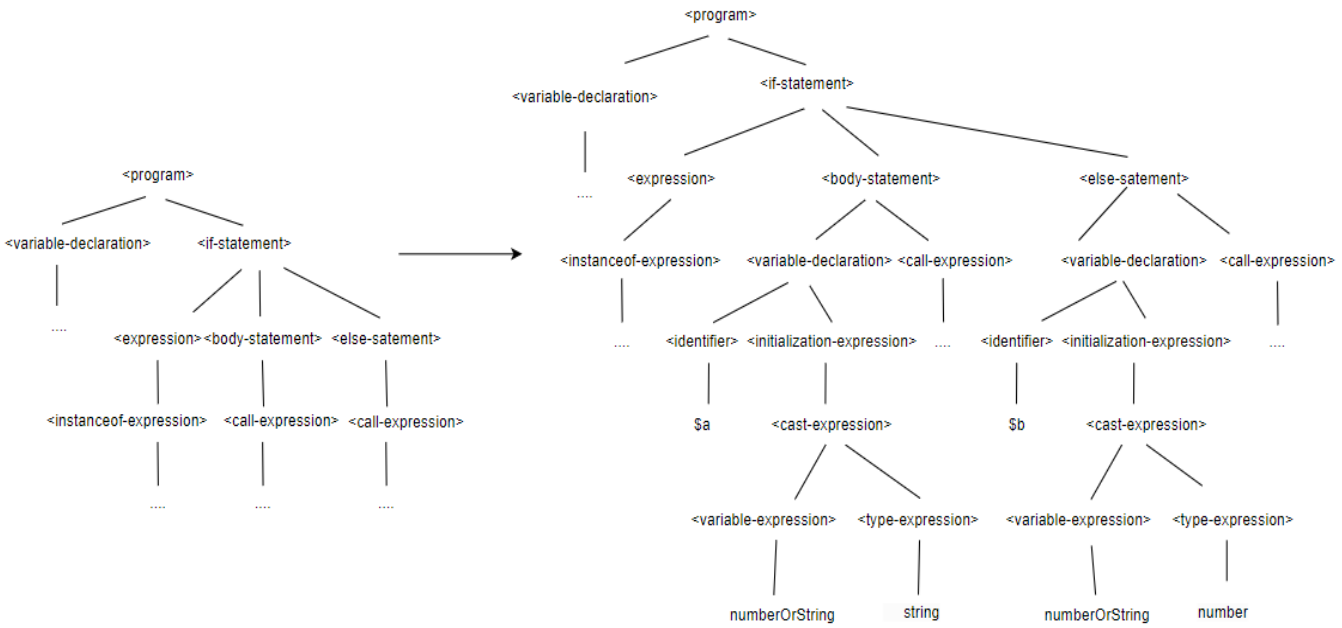

Рисунок 2 – Приклад перетворення дерева розбору за описаним способом введення додаткових змінних з певним типом

На останньому етапі необхідно визначити, у який тип віртуальної машини CLR платформи .NET необхідно транслювати оголошення змінної, результат функції, поля класу, тощо, які мають тип об'єднання. Звісно можемо виконати трансляцію у тип System.Object який є базовим для всіх типів і виконувати усі перевірки суто на рівні компілятора, який розроблюємо, але головним недоліком такого підходу є втрата інформації про можливі типи змінної на рівні збірки і при підключенні такої збірки до проектів, які написані на інших мовах програмування, що підтримують трансляцію у мову CIL (приклад C# чи Visual Basic), але не підтримують роботу з типами об'єднання, буде втрачена строга типізація.

Для того, щоб усунути цей недолік, в рамках запропонованого методу трансляції пропонується ввести додатковий узагальнений тип TSUntion, який приймає у вигляді аргументу можливі типи змінної.

```
public readonly struct TSUnion<T0, T1>
  {
     readonly T0 _value0;
     readonly T1 _value1;
     readonly int _index;
     TSUnion(int index, T0 value0 = default, T1 value1 = default)
     {
        _index = index;
        _value0 = value0;
        _value1 = value1;
     }
     public object Value =>
        _index switch
        {
           0 => _value0,
           1 => _value1,
           _ => throw new InvalidOperationException()
        };
     public int Index => _index;
     public bool IsT0 => _index == 0;
     public bool IsT1 => _index == 1;

     public T0 AsT0 =>
        _index == 0 ?
```

```
            _value0 :
            throw new InvalidOperationException($"Cannot return as T0 as result is T{_index}");
        public T1 AsT1 =>
            _index == 1 ?
                _value1 :
                throw new InvalidOperationException($"Cannot return as T1 as result is T{_index}");

        public static implicit operator TSUnion<T0, T1>(T0 t) => new TSUnion<T0, T1>(0, value0: t);
        public static implicit operator TSUnion<T0, T1>(T1 t) => new TSUnion<T0, T1>(1, value1: t);

        bool Equals(TSUnion<T0, T1> other) =>
            _index == other._index &&
            _index switch
            {
                0 => Equals(_value0, other._value0),
                1 => Equals(_value1, other._value1),
                _ => false
            };
        public override bool Equals(object obj)
        {
            if (ReferenceEquals(null, obj))
            {
                return false;
            }
            return obj is TSUnion<T0, T1> o && Equals(o);
        }
        public override string ToString() =>
            _index switch
            {
                0 => _value0.ToString(),
                1 => _value1.ToString(),
                _ => throw new InvalidOperationException("Unexpected index.")
            };
        public override int GetHashCode()
        {
            unchecked
            {
                int hashCode = _index switch
                {
                    0 => _value0?.GetHashCode(),
                    1 => _value1?.GetHashCode(),
                    _ => 0
                } ?? 0;
                return (hashCode * 397) ^ _index;
            }
        }
    }
```

Рисунок 3 – Лістинг класу TSUntion на мові програмування C# для типу об'єднання, що може мати 2 можливих типи

До переваг даного методу можна віднести:

- наявність у збірці (що є результатом виконання компіляції програми) інформації про усі можливі типи для типу об'єднання;
- змінні типу об'єднання матимуть додаткові методи для роботи з ними, що полегшує написання програм.

До недоліків даного підходу можемо віднести те, що у платформі .NET існує обмеження на максимальну кількість аргументів, які можуть бути передані в узагальнений тип. Виходячи з цього, визначимо тип TSUnion для випадків, коли тип об'єднання має від 2 до 9 можливих типів. Для наочності на рисунку 3 зображено лістинг узагальненого класу на мові програмування C# для типів об'єднання двох типів.

Кожний клас має окреме поле для зберігання значення певного типу, а також деякі методи, що полегшують роботу з цим типом даних. Перевизначені оператори приведення типу для типів, які змінна може мати, полегшують роботу з приведення типів примітивів до типів об'єднання. Перевизначені методи порівняння (Equals), хешування (GetHashCode) та перетворення у рядок (ToString) покращують роботу з базовими бібліотеками платформи .NET. Методи IsT та AsT полегшують перевірку і приведення до певного типу. Саме ці методи є ключовими при компіляції оператора isntanceof. Ці методи пронумеровані певним числом від 0 до 9, яке відповідає порядковому номеру типу серед

аргументів узагальненого класу.

Виконаємо компіляцію цих класів у збірку TS.Infrastructure.dll та додамо підтримку підключення цієї збірки до результуючої програми, що дозволить використовувати тип TSUnion при генерації IL інструкцій компілятором, що розроблюється.

Наступні дії запропонованого методу полягають у виконанні компіляції проміжного представлення програми, яка містить типи об'єднання, за такими правилами:

1. Оголошення змінної, поля класу чи результату функції, що мають тип об'єднання, транслюємо у тип TSUnion з відповідною кількістю аргументів.
2. Оператор instanceof для змінних типів об'єднання транслюємо у перевірку на певний тип, використовуючи відповідний метод IsT, до типу, на який виконується перевірка типу.
3. Компіляція приведення до необхідного типу виконується використовуючи відповідний необхідному типу метод AsT класу TSUnion.

Результат компіляції тестової програми (рисунок 1) за запропонованим методом у IL-інструкції зображено на рисунку 4.

```
    .entrypoint
    .maxstack 2
    .locals init (
      [0] valuetype T.TSUnion`2<int32, string> numberOrString,
      [1] string a1,
      [2] int32 b1
    )
    IL_0000: ldc.i4.s     10 // 0x0a
    IL_0002: call         valuetype TS.Infrastructure.TSUnion`2<!0/*int32*/, !1/*string*/> valuetype
ConsoleApp3.TSUnion`2<int32, string>::op_Implicit(!0/*int32*/)
    IL_0007: stloc.0      // numberOrString
    IL_0008: ldloca.s     numberOrString
    IL_000a: call         instance bool valuetype TS.Infrastructure.TSUnion`2<int32, string>::get_IsT1()
    IL_000f: brfalse.s    IL_0025
    // [7 5 - 7 34]
    IL_0011: ldloca.s     numberOrString
    IL_0013: call         instance !1/*string*/ valuetype TS.Infrastructure.TSUnion`2<int32,
string>::get_AsT1()
    IL_0018: stloc.1      // a1
    IL_0019: ldstr        "This is string: "
    IL_001e: ldloc.1      // a1
    IL_001f: call         void [System.Console]System.Console::WriteLine(string, object)
    IL_0024: ret
    IL_0025: ldloca.s     numberOrString
    IL_0027: call         instance !0/*int32*/ valuetype TS.Infrastructure.TSUnion`2<int32,
string>::get_AsT0()
    IL_002c: stloc.2      // b1
    IL_002d: ldstr        "This is number: "
    IL_0032: ldloc.2      // b1
    IL_0033: box          [System.Runtime]System.Int32
    IL_0038: call         void [System.Console]System.Console::WriteLine(string, object)
    IL_003d: ret
  } // end of method Program::'<Main>$'
```

Рисунок 4 – Лістинг на мові CIL скомпільованої тестової програми за запропонованим способом

Аналіз отриманих результатів.

Через відсутність аналогів компілятору мови TypeScript у проміжну мову CIL, які б підтримували трансляцію типів об'єднання, в якості аналога для аналізу результатів дослідження було вирішено обрати компілятор функціональної мови програмування F#, який підтримує схожу функціональність. Таким чином для тестування були написані 2 аналогічні програми на мовах програмування TypeScript та F#, що зображені на рисунку 5.

```typescript
for (let i = 0; i < 1000000; i++)
{
    const numberOrString: number | string = i % 2 ? 'test'
    const result = numberOrString instanceof number
        ? numberOrString + i
        : numberOrString.length + i
}
```

```fsharp
type NumberOrString =
    | Number of int
    | String of string

for i in 0..1000000 do
    let numberOrString = if i % 2 = 0
        then Number i
        else String "test"
    let result = match numberOrString with
        | Number num -> num + i
        | String str -> str.Length + i
        |> ignore
```

Рисунок 5 – Лістинг програм на мовах TypeScript (зліва) та F# (справа), що тестувались

s

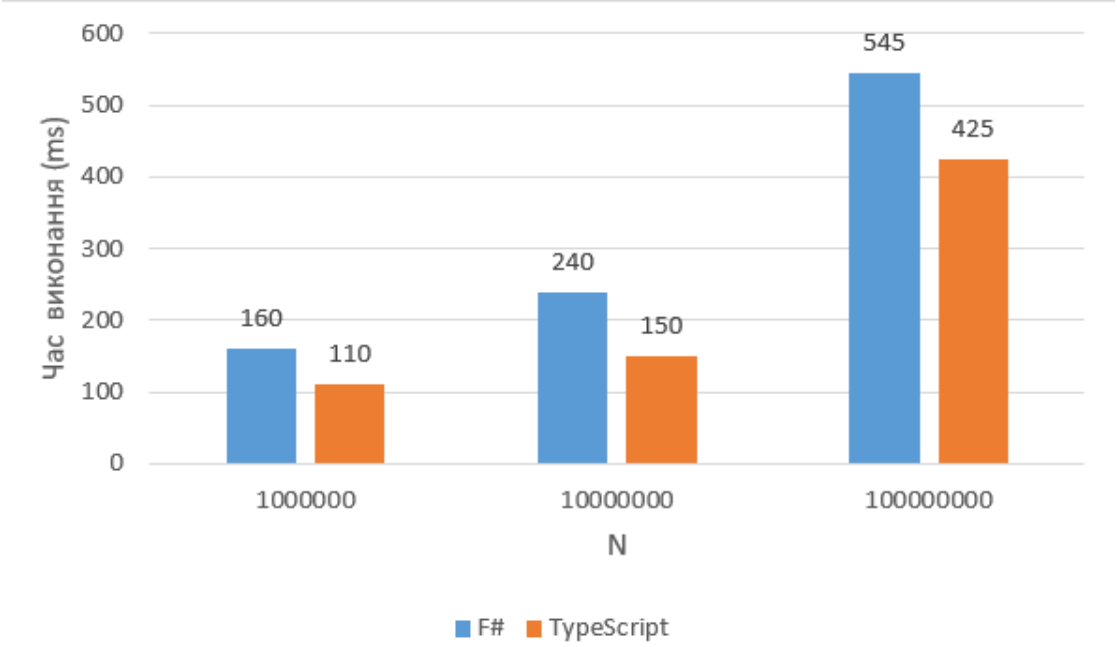

Рисунок 6 – Графік порівняння швидкодії згенерованих інструкцій для типів об'єднання компілятором F# та компілятором TypeScript за запропонованим способом.

Висновки.

При продовженні дослідження методів компіляції мови TypeScript у проміжну мову CIL платформи .NET було розроблено метод компіляції типів об'єднання шляхом введення спеціального узагальненого класу TSUnion, який зберігає інформацію про усі можливі типи, а також має спеціальні методи, що спрощують роботу з ним як під час трансляції, так і під час роботи зі скомпільованою бібліотекою на мові TypeScript при використанні її у програмах, написаних на інших мовах програмування платформи .NET.

В результаті реалізації у тестовому компіляторі описаного у цій статті способу, був отриманий результат, аналіз якого продемонстрував кращу у 1.3 – 1.6 разів швидкодію згенерованого коду розробленим тестовим компілятором, ніж код, згенерований компілятором-аналогом, в якості якого був взятий компілятор мови F#. Цей компілятор було обрано аналогом через відсутність інших компіляторів

мови TypeScript у мову CIL, що мають схожі функціональні можливості.

Отже, враховуючи отримані результати, а також результати минулих досліджень [3,4,5], можна ще раз підтвердити доцільність створення прямого компілятора мови програмування TypeScript у проміжний код CIL платформи .NET, а описане 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 [Електроний ресурс] – Режим доступу: https://cit-journal.com.ua/index.php/cit/article/view/352/451
4. Іваненко А.Р., Марченко О.І Спосіб компіляції замикання вкладених функцій мови TypeScript у проміжну мову CIL платформи .NET [Електроний ресурс] – Режим доступу: http://cit-journal.com.ua/index.php/cit/article/view/233/322
5. Марченко О.І., Іваненко А.Р. Аналіз способів трансляції мови TypeScript у проміжну мову CIL платформи .NET платформи .NET.
6. Марченко О.І. Основи проектування трансляторів: Конспект лекцій, Київ 2021 [Електронний ресурс] – Режим доступу: https://ela.kpi.ua/handle/123456789/45710
7. Рихтер Дж. CLR via C#. Программирование на платформе Microsoft .NET Framework 4.5 на языке C#. 4-е изд., СПб.:Питер, 2013
8. Документація TypeScript [Електронний ресурс] – Режим доступу: https://www.typescriptlang.org/docs/handbook/unions-and-intersections.html.
9. 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. Ivanenko A.R., Marchenko O.I Compilation the closure of TypeScript nested function into Common Intermediate Language of .NET platform [Electronic resource] – Access mode: https://cit-journal.com.ua/index.php/cit/article/view/352/451
5. Marchenko O.I., Ivanenko A.R. Analysis of TypeScript translation methods into Common Intermediate Language of .NET platform
6. Marchenko O.I. Fundamentals of translator design., Kyiv 2021 [Electronic resource] – Access mode: https://ela.kpi.ua/handle/123456789/45710.
7. Jeffrey Richter. CLR via C#. 2012
8. TypeScript documentation [Electronic resource] – Режим доступу: https://www.typescriptlang.org/docs/handbook/unions-and-intersections.html
9. Mono.Cecil documentation [Electronic resource] – Access mode: https://cecil.pe/