Міністерство освіти і науки України

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»

JIT компіляція динамічних мов програмування

Текстова частина до курсової роботи
за спеціальністю „Програмна інженерія” 121

Керівник **курсової роботи**
д.т.н., доц. Глибовець Андрій Миколайович

_____
(підпис)
“____” _____ 2021 р.

Виконав студент
Петрик Ярослав Ігорович
“____” _____ 2021 р.

Київ 2022

Міністерство освіти і науки України

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»

ЗАТВЕРДЖУЮ
д.т.н., доц. Глибовець Андрій Миколайович
_____
(підпис)
„____”_____ 2022 р.

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ
на курсову роботу

студенту 4-го курсу, факультету інформатики Петрик Ярославу Ігоровичу

ТЕМА: JIT компіляція динамічних мов програмування

Вихідні данні:

Результати впливу різних імплементацій інтерпретації динамічних програм

Зміст текстової частини до курсової роботи:

Індивідуальне завдання
Анотація
Вступ
1. Особливості динамічних мов
2. Процес виконання коду
3. Статичний аналіз
4. Динамічний аналіз
Висновки
Список літератури

Дата видачі „___” _____ 2021 р.

Керівник (підпис)

Календарний план виконання курсової роботи

Тема: JIT компіляція динамічних мов програмування

Календарний план виконання роботи:

№ п/п	Назва етапу дипломного проекту (роботи)	Термін виконання етапу	Примітка
1.	Отримання завдання на дипломну роботу	4.10.2021	
2.	Огляд технічної літератури за темою роботи	10.12.2021	
3.	Початкова розробка додатку	20.12.2021	
4.	Розробка методики збирання даних	5.02.2021	
5.	Імплементація байткоду та віртуальної машини	12.03.2021	
6.	Збір та аналіз даних	14.04.2011	
7.	Початок написання текстової частини	24.04.2020	
8.	Корегування роботи за уточненнями наукового керівника	18.05.2022	
9.	Захист курсової роботи	27.05.2022	

Студент: **Петрик Ярослав Ігорович**

Керівник: **Глибовець Андрій Миколайович**

“_____”_____

Зміст

Анотація

Метою дослідження є огляд та аналіз методів виконання, аналізу, та компіляції динамічних мов програмування. Оглянуто процес перетворень та оптимізації від текстового формату, до різних проміжних репрезентацій обчислень.

Для аналізу ефективності перетворень коду, та його інтерпретації, було створено проект luar — JIT компілятор мови Lua.

Ключові слова: JIT, Дизайн компіляторів, Динамічні мови програмування, JVM, Bytecode, Lua, Static Single Assignment.

Перелік прийнятих скорочень

JVM – Java Virtual Machine

JIT – Just-in-Time

АСД – Абстрактне Синтаксичне Дерево

WASM – WebAssembly

ОС – Операційна система

VM – Virtual Machine

CPS – Continuation Passing Style

SSA – Static Single Assignment

PGO – Profiler-Guided Optimization

Вступ

Ще з часів мови LISP, динамічні мови програмування користуються значною популярністю. Це і не дивно, адже використання подібних середовищ надає суттєві переваги на відміну від своїх статично типізованих еквівалентів:

- Швидкий час прототипування (Відсутність кроку попередньої компіляції)
- Можливість підміни будь-яких типів даних, та поведінок, під час виконання
- Спрощений синтаксис (Відсутність нотації специфікації типів)
- Портативністю (Виконуваний код зазвичай є текстом, або байткод репрезентацією, що не залежить від конкретної архітектури машини)
- Динамічною інтроспекцією (Можливість дослідити та маніпулювати будь-який комплексний тип даних під час виконання)

Хочу помітити, що ці переваги не є ексклюзивними для сімейства динамічних мов програмування. Наприклад мова Java хоч і є статично типізованою, проте має перелік динамічних аспектів, більшості з яких вона завдячує динамічній натурі JVM байткоду, в який та компілюється. Надбудови над динамічною мовою, як наприклад TypeScript над мовою JavaScript, також надає статичну типізацію, не втрачаючи динамізм середовищ виконання JavaScript.

Відсутність попередньої компіляції, портативний власний формат та динамічні можливості віддаляють модель програмування у таких мовах від заданою фактичними реальними обчислюваними машинами. Багато високорівневих аспектів таких мов не мають аналогів у реальному залізі, а тому потребують додаткових інструкцій для їх реалізації. Це робить динамічні аспекти цих мов складними, а отже і повільними в виконанні.

Потреба до швидкодії таких систем, різко зростає. Розробники прагнуть підвищити швидкість виконання застосунків, не жертвуючи динамізмом, що надають ці середовища.

Методики покращення виконання коду динамічних мов і є метою мого дослідження, а саме аспекти так званої Just-In-Time (JIT) компіляції.

1. Особливості динамічних мов

1.1. Можливості динамічних систем

Традиційно динамічні мови програмування мають додаткові можливості порівняно із статичними аналогами. Багато наведених можливостей є спільними як для мов, виконання яких представлені у текстовому форматі, так і у бінарній проміжній репрезентації — байткоді[1]. Проте деякі є унікальними для систем, що використовують написаний автором, високорівневий, сприйнятним для читання, код у текстовому форматі.

1.1.1. Відсутність кроку попередньої компіляції

Ця особливість більш властива мовам, що описують виконуваний код у текстовому форматі, сприйнятним для читання, та написання людьми. Це скорочує час затрачений на компіляцію, і переносить цей тягар в рантайм[1]. Відсутність цього кроку дозволяє пришвидшити час прототипування.

Багато динамічних систем використовують проміжну репрезентацію — байткод[1]. В такому випадку процес компіляції все ще присутній, проте зазвичай, більш спрощений у порівнянні із мовами що компілюються відразу в машинний код. Такий підхід дозволяє застосувати додаткові можливості мови (як наприклад статична типізація мови Java) та отримати більшість переваг динамічних мов під час виконання.

1.1.2. Можливість підміни будь-яких типів даних та поведінок під час виконання

Код динамічних систем поліморфний за замовчанням. Один і той самий код, може оброблювати данні різного вигляду і формату. Поведінка різних типів даних може бути зазначена у описі цих структур.

---

[1] Рантайм — програмне забезпечення, що відповідає за виконання користувацького коду

1.1.3. Можливість динамічного розширення функціоналу

Такі динамічні системи зазвичай мають можливість додавати новий, чи модифікувати існуючий код під час виконання програми. Така можливість є дуже бажаною для великої кількості систем, як наприклад:

- Браузер має потребу підвантажувати виконуваний код, із мережі інтернет, за потреби, щоб надати різноманітний функціонал різних вебсторінок.
- Віртуальні машини, як наприклад BEAM чи JVM, отримують змогу динамічно підмінювати код та проводити підтримку систем "на ходу", не зупиняючи поточні процеси.
- Автономні космічні системи, мають мати можливість оновлення програмного забезпечення, без втручання оператора[2], при цьому не порушуючи поточну роботу системи, адже такі системи мають функціонувати без зупинки та повернення на землю довгі роки.

1.1.4. Спрощений синтаксис

Динамічні мови часто обирають відмовлятись від нотації типів, що потрібні для генерації ефективного машинного коду, у користь інших динамічних можливостей, як наприклад, підміна типів даних, динамічне розширення, та інтроспекція. Саме тому програмний код та рантайм зазвичай не може бути впевненим у конкретному представленні даних та функцій у будь-який час виконання.

1.1.5. Портативність

Через те, що виконання коду динамічних систем не є можливим на актуальному залізі без модифікацій та трансляцій, такі системи є незалежними від конкретної архітектури машини, на якій код буде виконуватись. Репрезентації, що використовують такі системи, є більш високорівневими, а тому мають більшу вірогідність

бути виконаними. Низькорівневі набори інструкцій часто різняться одна від одної, проте високорівневі обчислювальні можливості майже повністю еквівалентні.

1.1.6. Динамічна інтроспекція

Статично-компільовані системи можуть уминати високорівневий опис структур даних, адже код згенерований для роботи над ними, старанно оптимізований, і має знання про їх вигляд, що гарантує бути сталим.

Система не може знати яку саме поведінку вона має застосувати до якого набору даних. Через велику кількість невідомих змінних, під час виконання програми, динамічні системи потребують кодувати більше інформації, що описують поведінку та вигляд об’єктів, під час виконання. Доступ до цього закодованого опису об’єктів системи, надається коду, який може вирішувати яку поведінку застосовувати надалі, та яким чином використовувати надану інформацію.

1.2. Проблеми виконання динамічних мов

Основна проблема ефективного виконання динамічних мов полягає у нестатку інформації про виконання. Брак інформації "на перед", означає відтермінування рішень, щодо виконання програми, до насамперед настання самого моменту інтерпретації коду. Система не може обробити ці рішення заздалегідь, а тому вимушена динамічно вирішувати подальший ефект команди.

Ця особливість є ключовою у всіх аспектах дизайну програмного забезпечення, його виконання та оптимізації. Відкладення рішень до останнього моменту надає можливість динамічно змінювати поведінку системи під час її роботи.

Проте, на практиці абсолютна більшість цих рішень є або статично незмінними, або лишаються сталими значний проміжок часу роботи системи. Система вимушена робити зайві обчислення, песимістично налаштована на те, що зміна відбудеться у будь-який момент. Хоча у більшості випадків, зміна так і не настає.

Як буде наведено пізніше у дослідженні, сучасні системи застосовують попередній аналіз, задля того щоб усунути резолюцію цих рішень із безпосередньо самого процесу виконання. Цей аналіз, як і у випадку попередньої компіляції, може займати суттєву кількість процесорного часу та використаної пам’яті. Тому до проблематики виконання додається ще затрачені ресурси попереднього аналізу коду динамічних систем.

2. Процес виконання коду

В цій роботі було досліджено процес виконання динамічного коду на прикладі мови програмування Lua. Lua — доволі проста, динамічна мова програмування, розроблена для інтеграції у існуючі системи, задля їх легкого розширення та кастомізації. Lua зазвичай виконується відразу із текстового формату, без компіляції у байткод репрезентацію. Тому буде проведено аналіз процесу виконання коду, оригінально представленого у текстовому форматі.

2.1. Обробка тексту (лексифікація)

Вхідним форматом для досліджуваного коду є текст. Проте цей формат оптимізований для сприйняття людиною. Він містить широку кількість інформації, що не несе жодного впливу на виконання програми, як наприклад форматування (відступи, переноси на нові рядки і.т.п.) та коментарі. Нашою першочерговою задачею є фільтрація та розбиття вхідного потоку даних на лексеми: послідовність семантично-важливих ключових слів, символів, ідентифікаторів та літералів.

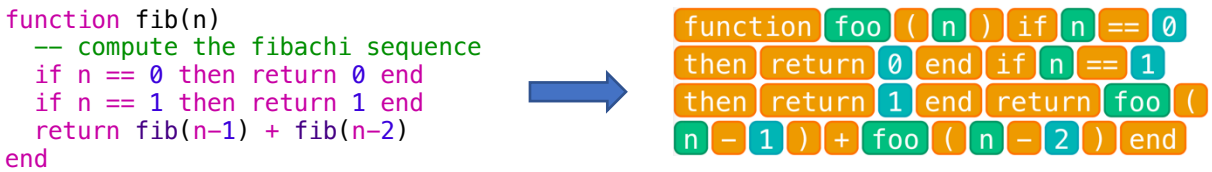


Рисунок 1.2.1: Лексифікація тексту

2.2. Парсинг

Ефективно працювати із потоком фонем все ще неможливо. Ми звикли описувати текстом структури обчислень, такі як: функції, арифметичні операції, присвоєння, тощо. Цієї структури нам бракує для ефективного опису програми.

Процес парсингу мов програмування детально розглянуто не буде, адже це доволі відома, добре досліджена сфера комп'ютерних наук, що не складає основного інтересу дослідження. Проте результат цієї операції вкрай важливий для подальших кроків виконання. Варто також зазначити, що так як кроку попередньої компі-

ляції не існує, час та ресурси затрачені на обробку мови напряму впливають на ефективність виконання коду, адже ця частина є невід'ємною у його інтерпретації.

Результатом цієї операції є структура даних, що називається Абстракте Синтаксичне Дерево (АСД). Це деревовидна структура, що кодує синтаксичні структури мови. Кожен вузол напряму визначає всі конструкції описані в текстовому представленні, такі як: оператори розгалуження, цикли, виклики функцій, операторів повернення, тощо.

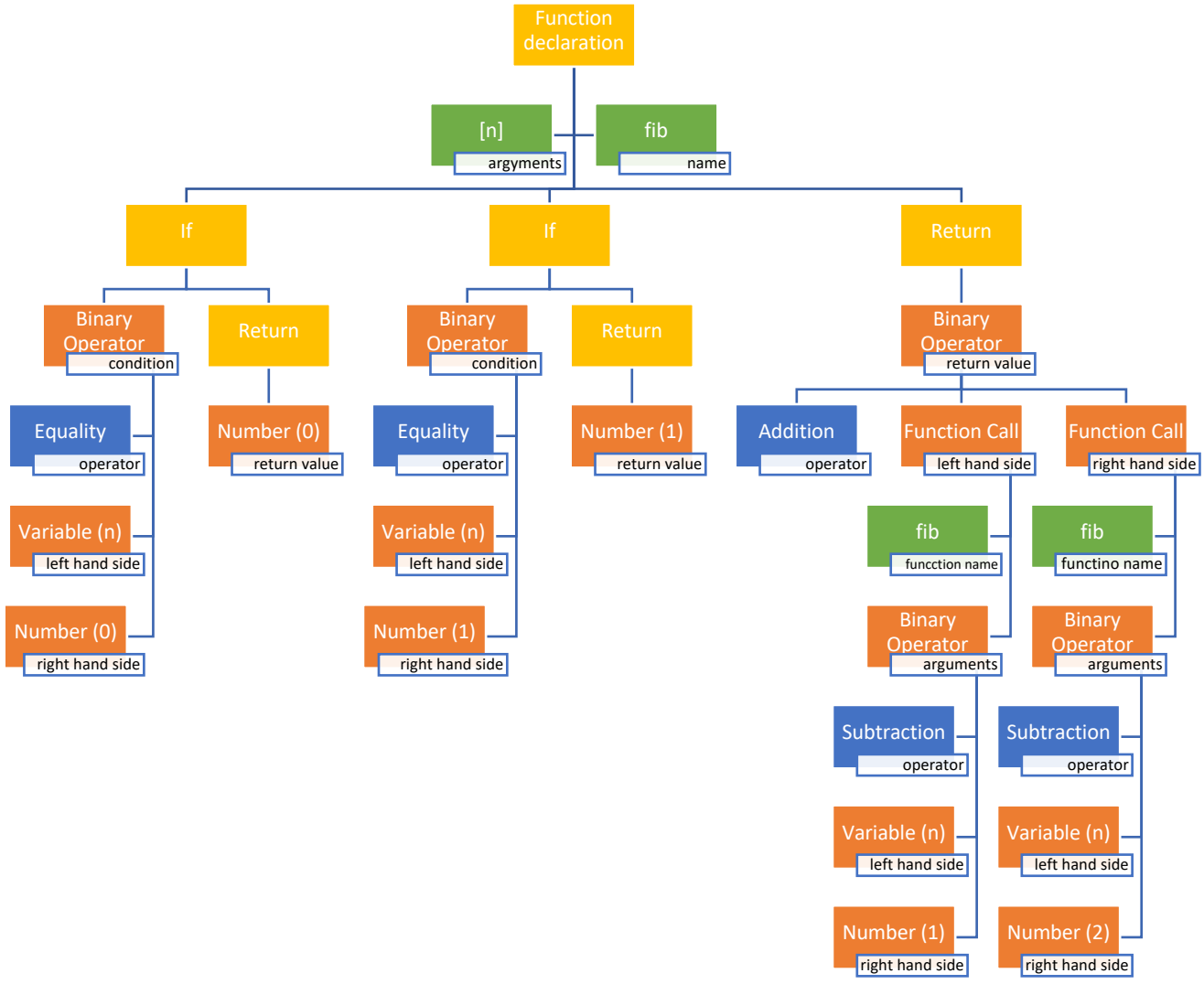


Рисунок 2.2: Приклад абстрактного синтаксичного дерева

Хочу зауважити, що цей процес зворотній — синтаксичне дерево можна перетворити на потік лексем, що його породжують. Цей аспект важливий для форму-

вання подальших зручностей для програмістів, наприклад: покращений вивід помилок, форматування коду, зовнішній статичний аналіз, трансформації коду, тощо.

2.3. Виконання Абстрактного Синтаксичного Дерева

АСД вже надає достатньо інформації щоб закодувати обчислення, та дозволити ефективну його інтерпретацію. Обхід цього дерева є чи не найпростішим способом виконання коду[3]. Маючи опис роботи структур мови, можна тривіально скласти інтерпретацію цієї програми.

Інтерприктація АСД є доволі легкою в реалізації. Вона кодує семантику мови так само, як це описано у вхідному коді. Складання АСД відносно швидкий процес, що виконується лише одноразово, при завантаженні програми.

Проте такий підхід має і низку недоліків:

2.3.1. Фрагментоване та розсіяне представлення

АСД дерево може бути необмежене у глибині. Вузли дерева, що описують структуру мови, часто рекурсивні, що призводить до потреби виділення даних на купі. У свою чергу, це призводить до неефективної використання пам’яті та кешу сучасних процесорів, що призводить до низької швидкодії[4].

2.3.2. Складність подальших оптимізацій

АСД це перша репрезентація, що дозволяє робити значні ітерації та оптимізації коду. Оптимізації пов’язані із обчисленням константних виразів, чи переписування вузлів у більш легкі для інтерпретації[5], доволі легко реалізувати використовуючи дану репрезентацію.

Проте існує безліч інших оптимізацій які виконати відносно цієї репрезентації доволі складно. Детальніше ці оптимізації буде розглянуто у наступному розділі.

2.4. Проміжна репрезентація

Задля спрощення виконуваного коду, можливо конвертувати існуюче АСД, до більш "пласкої" репрезентації. Тобто перейти від глибокого синтаксичного дерева, до одномірної послідовності інструкцій.

Такий перехід несе за собою втрату синтаксичних структур коду. Часто семантика мови втрачає сенс без існування таких структур. Проте навіть якщо проміжні кроки не відповідають структурам оригінальної мови, загальна спостережувана поведінка системи лишається сталою.

Втрата таких структур несе за собою потребу у створенні нових, більш низькорівневих. Можна виділити декілька популярних низькорівневих моделей обчислень, що використовують для проміжного опису коду:

2.4.1. Модель базована на стеку

В основі цієї моделі лежить набір інструкцій, що маніпулюють даними на стеку. Наприклад функція, що повертає суму 5 і 6 може виглядати наступним чином:

```
push 5
push 6
add
```

Рисунок 2.3: Приклад інструкцій машини базованої на стеку

Де перші дві команди додають константи нагору стеку, а команда add забирає два останніх числа із стеку, та повертає їх суму нагору. Компіляція коду, та інтерпретація такої моделі обчислень доволі легка. Прикладом систем, що виконують подібний код є JVM, .NET CLR та WebAssembly (WASM).

2.4.2. Модель базована на регістрах

В основі цієї моделі лежить набір інструкцій, що маніпулюють даними в регістрах. Їх кількість може бути як необмеженою, так і лімітованою.

Перевагою такого підходу, є зазвичай менша кількість інструкцій, для виконання еквівалентних обчислень, порівняно із моделлю на стеку. Такі інструкції також зазвичай описують більш низькорівневі операції, що може пришвидшити їх виконання[6]. Також набір команд зазвичай набагато більш схожий на той, що є рідним багатьом сучасним процесорам, тому легше транслюються в машинний код.

Недоліком такого підходу може стати більший розмір кожної інструкції, адже тепер вони мають містити адресу регістру, над яким відбуваються маніпуляції. Також компіляція може видатись більш складною, адже у разі компіляції до моделі обчислень, де є лімітована кількість регістрів, має проводитись нетривіальний аналіз їх використання. Прикладом систем, що використовують подібний підхід є:

- BEAM VM (віртуальна машина мови Erlang)
- Dalvik (віртуальна машина застосувань ОС Android)
- Lua VM
- JSCore (JavaScript та WebAssembly двигун WebKit браузерів)
- Spider Monkey (двигун браузера Firefox).

У досліджені було обрано саме цей стиль для представлення програм на Lua. Практичним шляхом знайдено приріст у більш ніж 2 рази порівняно із інтерпретацією АСД дерева напряму[21]. Тому надалі для демонстрації подальших оптимізацій буде використовуватись ця модель.

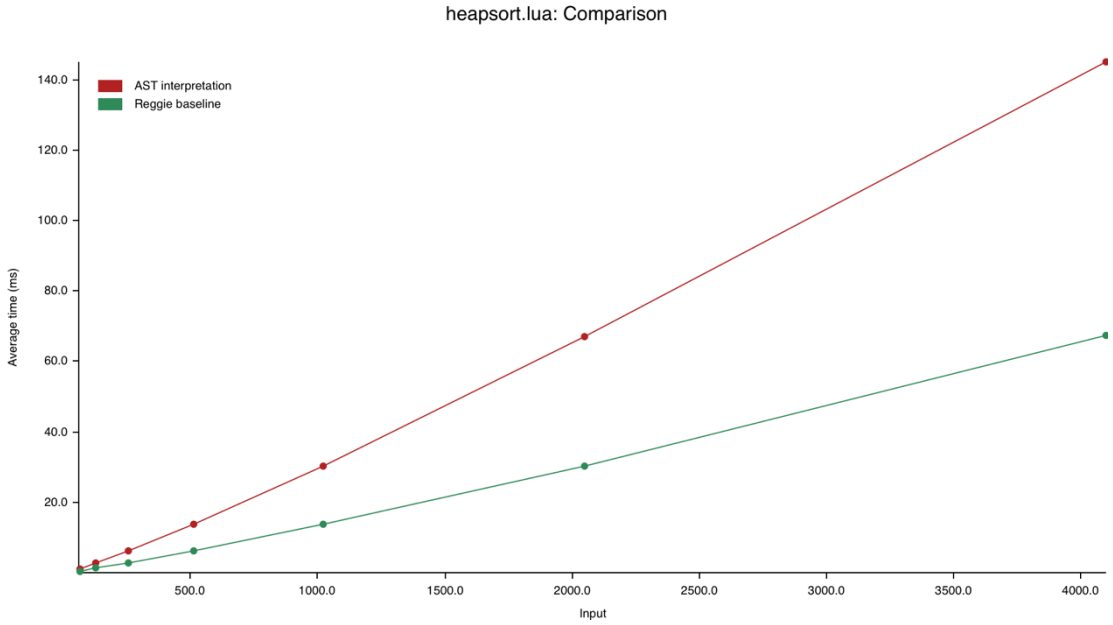


Рисунок 2.4: Порівняння швидкодії виконання АСД порівняно із репризентацією базованою на регістрах, на базі алгоритму Heapsort.

2.4.3. Continuation-passing style

Continuation-passing style[10] (стиль передачі продовжень, або CPS), більш властивий для використання функціональними мовами програмування. Такий підхід надає можливість легко застосовувати високорівневі оптимізації, наприклад аналіз лінивості обчислень.

Таке представлення обчислень базується цілком на лямбда-обчисленнях[7], вперше Алонзо Черчем у 1930-х роках.

2.5. Генерація машинного коду

На даний момент, найшвидшим інтерпретатором коду є мікропроцесор. У кожного програмуємого процесора є власний набір інструкцій, що він виконає якомога швидше. Тому останнім кроком у перетвореннях програм буде машинний код процесора. Ми кодуємо рішення про виконання кожної інструкції відразу до коду.

Нажаль цей крок є спеціалізованим для кожної архітектури обчислювальної машини, виконання на якій ми плануємо підтримувати.

3. Статичний аналіз

За своїм дизайном, динамічні мови надають обмежену інформацію щодо виконання програм. Вони не містять нотації типів, тобто типи даних можуть змінитись у будь-який момент. Відсутність цієї інформації ускладнює процес генерації ефективного коду. Проте, статично відома інформація, що закодована в мові, може бути експлуатована задля оптимізації коду.

Методики аналізу та оптимізації, описані в цьому розділі, схожі на ті, що використовуються в статичних мовах програмування. Тематика оптимізації коду є неймовірно обширною, та досі досліджуваною сферою комп'ютерних наук, тому опис їх більшості не є доцільним у поточному контексті. Тим не менш, увагу буде зосереджено на тих загальноприйнятих аспектах статичного аналізу, що доцільно застосовувати, і що мають значну цінність у контексті динамічних систем.

3.1. Оптимізація доступу до змінних

Більшість мов програмування використовують іменовані ідентифікатори, для позначення змінних. Це є природнім для сприйняття та написання людьми. Зазвичай, ці мови також диктують правила, до якої саме змінної звертається ім'я у будь-який визначений час. Обчислення цих правил при кожному доступі до змінної займає багато обчислювальних ресурсів, та потребує (зазвичай багатьох) асоціативних структур даних, щоб пов'язати текстове ім'я змінної до її значення. Збереження таких структур, також негативно впливає на використання пам'яті.

Проте асоціація імені до самої змінної зазвичай[8] є статично відомою у будь-якому сегменті коду. Саме це і надає змогу обчислити ці асоціації заздалегідь, та використовувати більш оптимізовану імплементацію їх доступу.

3.1.1. Лексично обмежені локальні змінні

Багато сучасних мов мають концепт локальних змінних, та їх зоною видимості. Наприклад, мова Lua, має визначення локальних змін завдяки ключовому слову local.

Аналіз лексичних меж локальних змінних доволі простий. Варто тримати асоціації між іменем змінної у певній частині коду, та її позицією у кадрі стеку виклику функції[9]. Таким чином ми можемо замінити доступ за іменем, на доступ за позицією у кадрі стеку.

```
condition = 1

function foo()
  local a = 5
  print(a)
  if condition then
    local a = 7
    print(a)
    b = 42
  end
end
```

```
condition = 1

function foo()
  local[0] = 5
  print(local[0])
  if condition then
    local[1] = 7
    print(local[1])
    b = 42
  end
end
```

Рисунок 3.1: Умовне перетворення доступу до локальних змінних

3.1.2. Глобальні змінні

Всі імена, що не зустрічаються в локальних лексичних межах, означають доступ до глобальної змінної. У Lua декларувати такі змінні не потрібно. Можна встановити асоціацію між іменами глобальних змін, та сталим індексом. Таким чином можна перетворити попередній приклад на:

```
condition = 1

function foo()
  local[0] = 5
  print(local[0])
  if condition then
    local[1] = 7
    print(local[1])
    b = 42
  end
end
```

```
global[0] = 1

global[1] = function()
  local[0] = 5
  print(local[0])
  if global[0] then
    local[2] = 7
    print(local[2])
    global[3] = 42
  end
end
```

Рисунок 3.2: Умовне перетворення доступу до глобальних змінних

3.2. SSA форма

Static Single Assignment[11] (одноразове статичне присвоєння, або SSA) форма — це така проміжна репризентація що містить присвоєння до змінної, лише один раз. Така форма дає змогу спростити оптимізацію коду, наприклад:

- Пропагація констант
- Пропагація можливих допустимих значень
- Прибирання невиконуваного коду
- Алокація регістрів
- тощо.

$$\begin{aligned} X &\leftarrow 1 \\ X &\leftarrow 3 \\ Y &\leftarrow X * 2 \\ X &\leftarrow 4 \\ Y &\leftarrow X * Y \end{aligned} \quad \Rightarrow \quad \begin{aligned} X_1 &\leftarrow 1 \\ X_2 &\leftarrow 3 \\ Y_1 &\leftarrow X_2 * 2 \\ X_3 &\leftarrow 4 \\ Y_2 &\leftarrow X_3 * Y_1 \end{aligned}$$

Рисунок 3.3: Умовне перетворення до SSA форми

Проте цей підхід не спрацює, якщо в коді існують розгалуження чи цикли, що модифікують значення змінної. Для вирішення цієї проблеми SSA форма містить ϕ-вузли (фі-вузли).

```
:entry                          :entry
  a = 1                           a_1 = 1
  if condition                    if condition
    then goto else                  then goto else
  a = 2                           a_2 = 2
  goto cont                       goto cont
:else                           :else
  a = 3                           a_3 = 3
:cont                           :cont
  print(a)                        a_4 = phi(a_2, a_3, entry, else)
                                  print(a_4)
```

Рисунок 3.4: Приклад використання ϕ-ноди

ϕ-вузол робить вибір між значенням змінної a_2 та a_3 в залежності від того, який останній блок виконувався. В цьому прикладі, якщо останній виконуваний

блок називався entry — то буде обрано значення a_2, якщо ж останній блок був else — a_3

Хоч SSA форма і може інтерпретуватись, зазвичай вона використовується задля для оптимізації, та перетворюється, або назад до оригінальної репрезентації, або до іншого виконуваного формату.

3.3. Пропагація типів

Як відомо, виконання динамічних мов несе за собою купу проблем, пов’язаних із нестачею інформації. Проте ми можемо експлуатувати відомі нам властивості деяких операцій, щоб дізнатись більше інформації про типи об’єктів у коді.

Як наприклад, операція додавання. Із специфікації мови Lua, нам відомо, що результатом операції додавання завжди буде число. Які б невідомі операнди, ми не отримували, в результаті отримаємо завжди відомий нам тип. Знаючи тип змінної, ми можемо генерувати більш оптимізований код, що не робить перевірок на типи, чи складні обрахунки мета-методів.

Це стосується як операторів, так і відомих нам функцій. Наприклад, функція стандартної бібліотеки tostring, яка, як бачимо із назви, завжди повертатиме стрічку.

3.4. Оптимізація виклику функцій

Часто в динамічних мовах програмування функції є значеннями першого класу. Це означає, що їх можна зберегти у змінних, таблицях, передати як аргумент, тощо. Ця особливість лежить в основі динамічного виконання коду. Виклики функцій є основою сучасного коду, тому їх прискорення суттєво відображається на характеристиках швидкодії.

Проте, можливість зберігати арбітрарні функції, різного порядку, різної арності[2], ускладнює їх ефективний виклик. Наприклад, у мові Lua, функція може отримати на вхід будь-яку кількість аргументів, будь-якого типу, та повернути будь-яку кількість значень, будь-якого вигляду.

Не всі виклики мають зводитись до спільної репрезентації. В деяких випадках можливо встановити типи функцій, та використати більш ефективну імплементацію їх виклику.

Шляхом пропагування типів даних, анотації нативних[3] функцій, та динамічного аналізу (що буде описаний у наступному розділі), можна дізнатись:

а) Яка саме функція буде викликана
б) Кількість аргументів та їх тип
в) Які типи значень та їх кількість повертатиме функція

Цей аналіз не завжди може встановити точні дані кожного виклику, проте інформація, що ми вилучили у процесі, може бути використана для покращення викликів. Тому в представленій реалізації, скомпільовані функції містять дві згенеровані репрезентації — одна для випадків коли кількість аргументів та повернутих значень невідома, інша — коли ця інформація доступна.

3.4.1. Анотація нативних функцій

Будь-яка мова чи то динамічна система, в результаті повинна мати змогу спілкуватись із операційною системою. Всі сучасні мови мають способи взаємодіяти із процедурами lingua franca[4] комп'ютерних мов — C.

---

[2] Арність — кількість аргументів функції

[3] Нативні функції — такі функції, імплементація яких визначена у, рідному для машини, коді

[4] Спільна мова

Дуже часто стандартна бібліотека в себе включає функції із нативного середовища. Поведінка цих функцій відома при їх оголошені. Часто статично відома кількість аргументів, їх вигляд, та повернуті значення. Ми можемо анотувати ці значення, для використання у подальшому статичному та динамічному аналізі коду.

Також, часто функції поводять себе по-різному в залежності від кількості чи вигляду аргументів. В багатьох випадках ефективніше було б використовувати різні імплементації в залежності від переданих аргументів. Саме для цих потреб, є корисним надати декілька імплементацій однієї функції. Це схоже на статичний поліморфізм[12] у таких мовах як C++ чи Java, проте визначення імплементації вирішується під час виконання програми, а не її компіляції.

3.4.2. Інлайнинг[5]

Найшвидший виклик функції — це той що ніколи не відбувся. Знаючи яка саме функція буде викликана, можна замінити виклик тілом функції. Така підміна не лише зменшує витрати на створення нового стек фрейму, а й дозволяє компілятору бачити "скрізь" виклики функцій. Маючи інформацію про стан даних до виклику, можна спрогнозувати який шлях виконання, та яка поведінка буде виконана.

Здавалось б, інлайнинг має вирішує купу проблем, і варто всі виклики функцій замінити на їх тіло. Нажаль не все так просто.

Починаючи із рекурсивних функцій. Рекурсивні чи взаємо-рекурсивні функції неможливо інлайнити, адже це створить нескінченну кількість коду.

По-друге, інлайнинг збільшує кількість генерованого коду. Функція що була викликана у двох місцях після інлайнингу, її тіло буде повторюватись, як мінімум, двічі. Збільшення розміру виконуваного коду несе за собою значні недоліки. По-

[5] Від англ. inline (вписати в одну стрічку) — процес заміни виклику функції її тілом

перше, це збільшення використовуваної пам’яті для збереження послідовності інструкцій, а по-друге, це призводить до збільшення часу очікування підсистеми пам’яті на підвантаження інструкцій. Як відомо, пам’ять це найповільніша частина сучасних процесорів.

Тому для інлайнингу використовують ретельно підібрані евристики[13], що вирішують чи варто інлайнити функцію, чи ні. Вгадати наперед результуючу швидкодію при застосуванні операції інлайнингу дуже складно, тому ці евристики включають у себе низку різних факторів.

3.4.3. Інтрінзики[6]

Може здатись що нативні функції — це найшвидший спосіб виконати будь-яку операцію. І в дійсності, в виключно інтерпретованій мові, складно досягти швидкості подібної до, наприклад, статично компільованої C процедури. Проте компілятор не може бачити скрізь нативні функції, таким чином, як він бачить скрізь функції описані в самій мові. Інлайнити нативні виклики не є ні резонним, ні безпечним.

Код нативної функції, так само як і будь-яка інша функція в динамічній мові, не може знати тип та кількість аргументів, що вона отримує. Проте навідміну від функцій мови, компілятор не може застосувати пропагацію типів, констант, чи можливих значень в код що скомпільовано заздалегідь. Переписати нативну функцію неймовірно складно.

Щоб покращити видимість компілятора у такі функції, можна надати їх реалізацію описаною у коді (чи проміжній репрезентації) динамічної мови. Таким чином компілятор отримує можливість за необхідності інлайнити нативні функції.

---

[6] Від англ. intrinsics — функціонал (інструкції) вбудований до середовища виконання

Зазвичай нативні функції надають можливості взаємодіяти із зовнішнім оточенням віртуальної машини, і часто такі функції неможливо виразити лише за допомогою наданих можливостей мови. В таких випадках віртуальна машина отри-мує додаткові інструкції (інтрінзики), що описують низькорівневу взаємодію таких функцій[14]. Використати ці інтрінзики зазвичай не є можливим із користуваць-кого коду напряму, тому лише визначені функції можуть бути замінені на байткод із їх використанням.

4. Динамічний аналіз

Як було визначено раніше, основною проблемою в ефективному виконанні коду динамічних мов є нестаток інформації щодо вигляду даних[16]. Динамічний аналіз має фокус на зборі інформації під час виконання програми, та використання її для подальшої оптимізації.

4.1. Збір інформації

Для отримання інформації щодо виконання коду, потрібно збирати статистику під час його роботи. Функції представляють природню одиницю для аналізу та оптимізації[15]. Для збору інформації, кожен початковий виклик функції зберігає ін-формацію щодо аргументів, із якими вона викликається. Їх тип, їх значення, тощо.

Також має сенс збирати інформацію про значення, які повертаються із викликаних функцій, чи вони змінюються, чи змінилась глобальна змінна відносно попереднього виклику, який шлях був обраний при розгалуженні потоку виконання.

4.2. Прийняття припущень

Маючи інформацію щодо виконання, ми можемо зробити припущення, про деякі особливості обчислень. Наприклад, якщо зібрана інформація із минулого кроку, повідомляє, що функція викликалась лише із цілочисленними значеннями, то ми можемо зробити припущення, що ця функція буде викликатись лише із цілочисленними значеннями і надалі. Якщо функція повертала лише стрічки до цього моменту, можемо зробити припущення, що стрічки будуть повертатись і надалі.

Це стосується не тільки аргументів функцій та повернутих значень. Схожі припущення ми можемо робити і щодо обраного шляху виконання при розгалуженнях, та про значення і типи даних глобальних змінних, тощо.

Використовуючи ці припущення як джерело істини, ми можемо застосувати той самий статичний аналіз що і описаний у минулому розділі. Результат статично-

го аналізу із використанням припущень, має набагато більше інформації, і як результат, породжує більш спеціалізований, більш ефективний код.

4.3. Де-оптимізація

Однак, наші припущення обрані у попередньому кроці, можуть виявитись хибними. Зміна у зовнішніх факторах, чи динамічна заміна/ін'єкція коду може спричинити виконання інший набір інструкцій, із іншими даними. В результаті, прийняття припущень порушують динамізм програми, спеціалізуючи її лише до однієї незмінної імплементації.

Важливо мати змогу відкинути попередні припущення, та повернути виконання до попередньої не спеціалізованої версії процедури. Цей процес називається де-оптимізація.

Щоб знайти відмінність між припущеннями та реальною поведінкою, потрібно робити перевірки на правдивість припущень перед виконанням спеціалізованого коду. Ці перевірки мають деякі накладні витрати, проте у порівнянні із виконанням неоптимізованого коду, ці втрати швидкодії не є суттєвими.

4.3.1. Повернення до виконання неоптимізованого коду

Значну проблему складає процес повернення до неоптимізованого коду. Якщо, припустимо, маємо наступні фрагменти неоптимізованого та спеціалізованого коду:

```
<неоптимізований>
const i32 5 ; зберегти цілочисленне число 5 у акумуляторі
wrap i32 ; перетворити ціле число у динамічне значення
add dyn r0 ; виконати додавання двох динамічних (невідомих) значень
call print_dyn ; викликати динамічну функцію із результатом попередньої операції

<спеціалізований до використання цілих чисел>
check i32 r0 ; перевірити що в регістрі r0 ціле число, якщо ні де-оптимізувати код
const i32 5 ; зберегти цілочисленне число 5 у акумуляторі
add i32 r0 ; додати два цілих числа
typed_call print_i32 ; викликати спеціалізовану версію функції, для цілого числа
```

Як діяти у разі порушення припущень посеред виконання функції? Можна спробувати виконати цю саму процедуру, але її неоптимізований відповідник заново. Проте це можливо лише якщо процедура чиста, детерміністична, та не має спостережуваних побічних ефектів.

Інший підхід складається в тому, щоб зберегти метадані асоційовані з інструкцією перевірки припущень, щоб встановити відповідність між поточним місцем виконання, та станом змінних при виконанні у неоптимізованій версії[17]. Цей підхід стає складнішим якщо код є компільованим до машинного коду, адже потрібно встановити місцезнаходження даних не лише у пам'яті, а ще й у процесорних регістрах.

4.4. Характеристики використання динамічного аналізу

Використання динамічного аналізу має потенціал породити набагато більш оптимізований код, у порівнянні із відповідним кодом статичних мов, адже бере до уваги не лише інформацію надану автором програми, а і реальною її поведінкою.

Такий аналіз має схожість із технікою Profiler-Guided Optimization[18] (оптимізації керовані профайлером, або PGO), де скомпільований код виконується в очікуваних умовах і фіксується інформація про виконання. Такий аналіз подібний до кроку збору інформації динамічного аналізу. Ця інформація потім використовується для повторної компіляції коду, але із урахуванням цього профілю.

Динамічний аналіз наближає здібності швидкодії динамічних мов до статично компільованих, проте досягти повної еквівалентності не можливо, через накладні витрати пов'язані із перевіркою припущень.

Також подібні оптимізації можливі лише при існуванні сталих поведінок у коді. Якщо ж поведінка постійно змінюється, то це призводить до частої деоптимізації, що вимагає додаткових витрат процесорного часу.

Варто зазначити, що система динамічного аналізу потребує додаткової пам’яті[19] для збереження декількох версій коду, та метаданих отриманих під час збору інформації.

Також динамічний аналіз потребує час на збір даних про виконання. Якщо цей час скоротити, то збільшується шанс того, що код потрібно буде де-оптимізувати, через неочікувану поведінку. А якщо час витрачений на збір інформації завеликий, то ми втрачаємо процесорний час, що міг би бути краще використаний у оптимізованому коді. Для вирішення цієї проблеми JIT системи часто застосовують різні ретельно підібрані евристики та відтермінують застосування більш агресивної оптимізації[20]. Більш жорсткі припущення відтермінують, до часу коли система матиме значну впевненість у сталості поведінки коду. Час затрачений на збір даних призводить до повільного старту програм. Тому JIT системам зазвичай потрібен час "розігнатись", та наблизитись до швидкодії статично компільованих застосунків.

Підсумок

Багато сучасних систем потребують широких динамічних можливостей. Популярність динамічних мов зростає, а разом і з цим, зростає і потреба у швидкодії систем, що їх використовують. Технології JIT компіляції мають за мету досягти цієї цілі.

Код проходить низку трансформацій: від текстової репрезентації, до виконуваного машинного коду. Кожен крок несе за собою покращення у швидкодії порівняно із попереднім його представленням.

Традиційно такі системи мали виконуватись у середовищі інтерпретатора, але шляхом експлуатації статично відомих даних про програму, та спостеріганням за її виконанням у середовищі, можна отримати достатньо інформації, щоб генерувати ефективний та оптимізований код.

Ці аспекти динамічної компіляції, наближають динамічні мови, до статично компільованих. Методики компіляції динамічного коду мають багато спільного із такими, що компілюють статичний, проте системи, що використовують динамічні мови в JIT середовищі, все ще мають низку недоліків порівняно із статичними, що компілюються заздалегідь.

- Час, що був затрачений на аналіз та компіляцію статичних мов, тепер затрачено насамперед під час виконання.
- Брак статичної інформації змушує затратити час на її збір під час роботи програми.
- Припущення, що були взяті за істинні під час динамічного аналізу потребують частої перевірки на сталість, що додає накладні витрати при виконанні.
- Зміна характеристик виконуваної програми спричиняє де-оптимізацію, що вкрай негативно впливає на показники.

- Статичний, та особливо динамічний аналіз потребує додаткових ресурсів пам’яті машини, для своєї ефективної роботи.
- Резонування про характеристики швидкодії стає більш важким, через складну (і часто недетерміністичну) поведінку рантайму та генерацію коду.

Проте всі сучасні аспекти JIT система надають змогу отримати конкурентноспроможну швидкодію динамічних мов, не жертвуючи динамічними особливостями, та високорівневою репрезентацією мови.

Список літератури

[1] Bytecode — Wikipedia. [Електронний ресурс] 18.05.2022. hhttps://en.wikipedia.org/wiki/Bytecode

[2] Hacking Apollo 14: How an MIT computer scientist saved a lunar landing. [Електронний ресурс] 18.05.2022. https://astronomy.com/news/2019/06/hacking-apollo-14-how-an-mit-computer-scientists-saved-a-lunar-landing

[3] Let's Build A Simple Interpreter. Part 7: Abstract Syntax Trees. [Електронний ресурс] 18.05.2022. https://ruslanspivak.com/lsbasi-part7

[4] Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 1: Basic Architecture. ст.43-64. 19.05.2022. https://cdrdv2.intel.com/v1/dl/getContent/671436

[5] Thomas Würthinger, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Doug Simon, Christian Wimmer: Self-optimizing AST interpreters. 10.2012.

[6] Yunhe Shi, David Gregg, Andrew Beatty, M. Anton Ertl: Virtual Machine Showdown: Stack Versus Registers. 12.06.2005. https://static.usenix.org/events/vee05/full_papers/p153-yunhe.pdf

[7] Alonzo Church: The calculi of lambda-conversion. 1941.

[8] with — JavaScript | MDN. [Електронний ресурс] 19.05.2022. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/with#performance_pro_contra

[9] Wikipedia — Call stack. [Електронний ресурс] 19.05.2022. https://en.wikipedia.org/wiki/Call_stack

[10] Wikipedia — Continuation-Passing style. [Електронний ресурс] 19.05.2022. https://en.wikipedia.org/wiki/Continuation-passing_style

[11] Appel, A: Static Single-Assignment Form. In Modern Compiler Implementation in ML (ст. 427-467). Cambridge: Cambridge University Press. 1997. doi:10.1017/CBO9780511811449.020

[12] Wikipedia — Function overloading. [Електронний ресурс] 19.05.2022. https://en.wikipedia.org/wiki/Function_overloading

[13] Theodoros Theodoridis, Tobias Grosser, Zhendong Su: Understanding and Exploiting Optimal Function Inlining. ASPLOS 2022: Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems. 02.2022.

[14] jdk8/jdk8/hotspot: src/share/vm/classfile/vmSymbols.hpp. [Електронний ресурс] 19.05.2022. http://hg.openjdk.java.net/jdk8/jdk8/hotspot/file/87ee5ee27509/src/share/vm/classfile/vmSymbols.hpp#l581

[15] Wikipedia — Inline Caching. [Електронний ресурс] 19.05.2022. https://en.wikipedia.org/wiki/Inline_caching

[16] Build me a JIT as fast as you can. [Електронний ресурс] 19.05.2022. http://www.mirandabanda.org/cogblog/2011/03/01/build-me-a-jit-as-fast-as-you-can/

[17] Fedor Indutny: Deoptimize me not, v8. [Електронний ресурс] 1.12.2014. https://darksi.de/a.deoptimize-me-not/

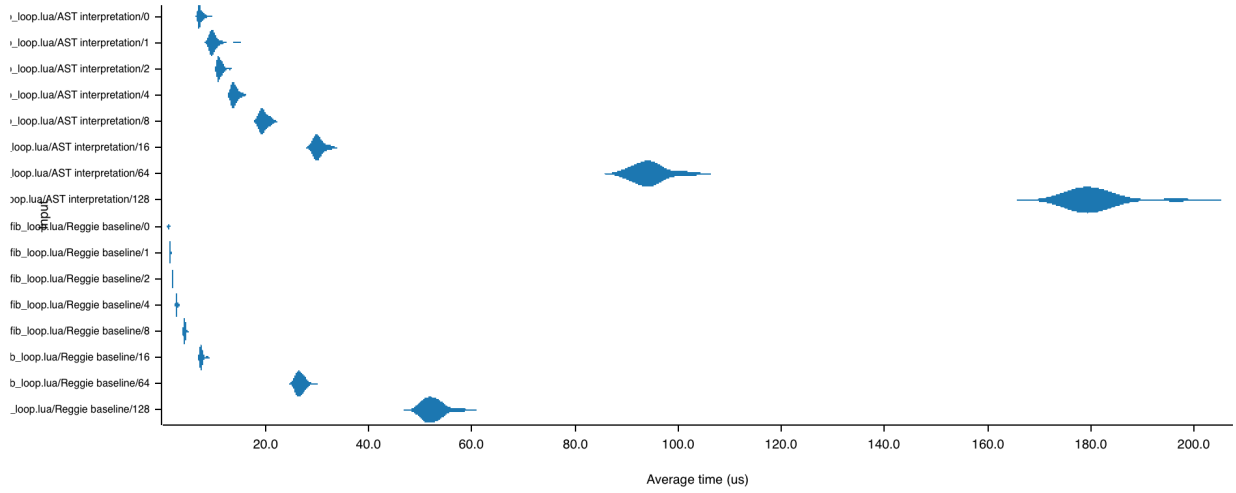
[18] Wikipedia — Profile-guided optimization. [Електронний ресурс] 19.05.2022. https://en.wikipedia.org/wiki/Profile-guided_optimization

[19] Mythri Alle, Dan Elphick, Ross McIlroy: A lighter V8. [Електронний ресурс] 12.09.2019. https://v8.dev/blog/v8-lite

[20] Filip Pizlo: Introducing the WebKit FTL JIT. [Електронний ресурс] 13.05.2014. https://webkit.org/blog/3362/introducing-the-webkit-ftl-jit/

[21] Yaroslav Petryk: Luar. [Електронний ресурс] 20.05.2022. https://github.com/Malien/luar

Додаток 1: Порівняння швидкодії коду різних середовищ виконання Lua коду, при обрахуванні послідовності Фібаначі.



AST interpretation / X — виконання АСД дерева напряму, для X-го числа послідовності

Reggie baseline / X— виконання проміжної репрезентації, базованої на віртуальній машині із регістрами, для X-го числа послідовності.

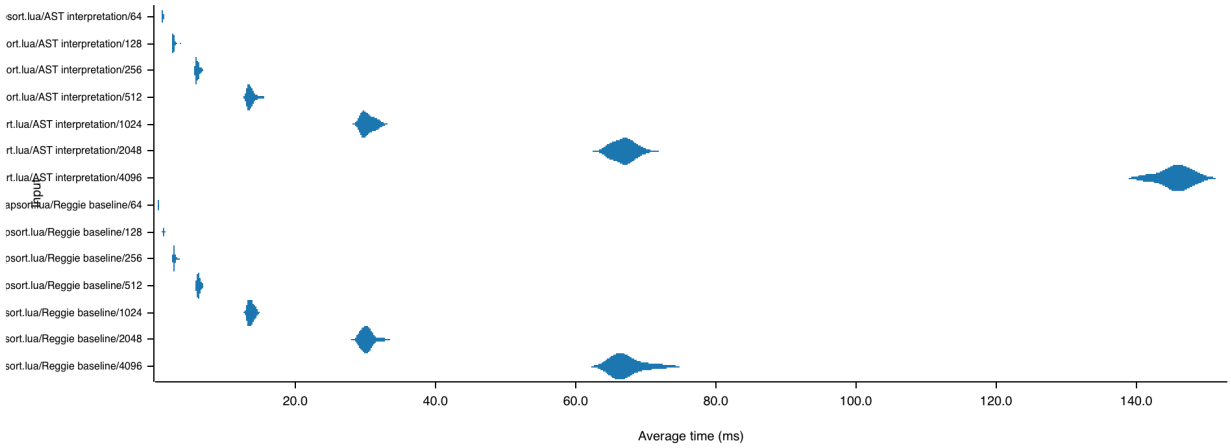
Статистичні показники швидкодії для АСД інтерпритації, для обрахування 16-го числа послідовності

	Lower bound	*Estimate*	*Upper bound*
Slope	29.632 us	29.897 us	30.173 us
R²	0.9751603	0.9788423	0.9748364
Mean	30.045 us	30.368 us	30.724 us
Std. Dev.	714.48 ns	1.0560 us	1.3174 us
Median	29.903 us	30.124 us	30.482 us
MAD	478.68 ns	880.33 ns	1.2064 us

Статистичні показники швидкодії для виконання проміжної репрезентації, базованої на віртуальній машині із регістрами, для обрахування 16-го числа послідовності

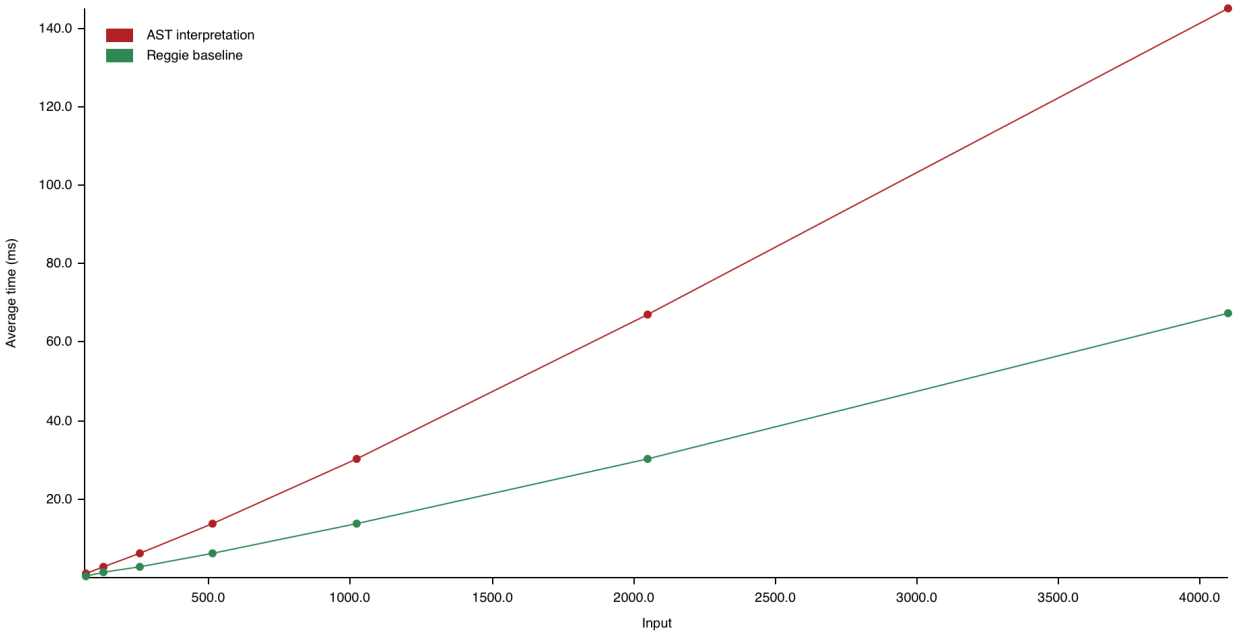
	Lower bound	*Estimate*	*Upper bound*
Slope	7.4481 us	7.5218 us	7.6030 us
R²	0.9645581	0.9689469	0.9636075
Mean	7.5372 us	7.6567 us	7.7964 us
Std. Dev.	203.09 ns	404.38 ns	542.08 ns
Median	7.4476 us	7.5226 us	7.6280 us
MAD	119.43 ns	207.95 ns	337.74 ns

Додаток 2: Порівняння швидкодії коду різних середовищ виконання Lua коду, при обрахуванні алгоритму Heapsort



AST interpretation / X — виконання АСД дерева напряму, для колекції розміром у X елементів.

Reggie baseline / X— виконання проміжної репрезентації, базованої на віртуальній машині із регістрами, для колекції розміром у X елементів.



Додаток 3: Статистичні данні швидкодії виконання алгоритму Heapsort у різних середовищах

Інтепритація АСД, для колекції розміром у 512 елементів.

	Lower bound	*Estimate*	*Upper bound*
R²	0.0200972	0.0213228	0.0198713
Mean	13.602 ms	13.730 ms	13.871 ms
Std. Dev.	365.86 us	520.60 us	652.35 us
Median	13.385 ms	13.500 ms	13.825 ms
MAD	122.81 us	352.98 us	645.48 us

виконання проміжної репрезентації, базованої на віртуальній машині із регістрами, для колекції розміром у 512 елементів.

	Lower bound	*Estimate*	*Upper bound*
R²	0.0160078	0.0170433	0.0159114
Mean	6.2403 ms	6.2988 ms	6.3601 ms
Std. Dev.	183.29 us	230.17 us	268.81 us
Median	6.2201 ms	6.3092 ms	6.3199 ms
MAD	92.760 us	291.48 us	354.38 us