DOI 10.37882/2223-2966.2024.1-2.03

ГЛОБАЛЬНЫЙ ЯЗЫК КАК КВИНТЕССЕНЦИЯ ИЗОМОРФИЗМА ЯЗЫКОВ ПРОГРАММИРОВАНИЯ

GLOBAL LANGUAGE AS THE QUINTESSENCE OF ISOMORPHISM OF PROGRAMMING LANGUAGES

A. Zheludkov
S. Grigoriev

Summary. This paper examines the concepts of isomorphism of algorithms and isomorphism of programming languages. Their connection is determined, namely, the implementation of the same algorithm is isomorphic in different paradigms and, as a consequence, languages. Thus, the isomorphism of algorithms turns into an isomorphism of programming languages, which means it is possible to create a global language with which it will be possible to convert code between existing languages and translate implementations of algorithms between different paradigms. The main features of this language are described.

Keywords: isomorphism of algorithms, equivalence of algorithms, programming paradigm, global programming language, Turing machine.

Желудков Антон Владимирович
Аспирант,
«Московский финансово-юридический университет»
zhantonv@gmail.com
Григорьев Сергей Георгиевич
Доктор технических наук, профессор,
чл.-корр. РАО, профессор, ГАОУ ВО «Московский городской педагогический университет»
grigorsg@yandex.ru

Аннотация. В данной работе рассмотрены понятия изоморфизма алгоритмов и изоморфизма языков программирования. Определена их связь, а именно реализация одного и того же алгоритма изоморфна в различных парадигмах и, как следствие, языках. Таким образом изоморфизм алгоритмов переходит в изоморфизм языков программирования, а значит существует возможность создания глобального языка, с помощью которого получится осуществлять конвертацию кода между существующими языками и выполнять перевод реализаций алгоритмов между различными парадигмами. Описываются основные особенности данного языка.

Ключевые слова: изоморфизм алгоритмов, эквивалентность алгоритмов, парадигма программирования, глобальный язык программирования, машина Тьюринга.

Введение

Утверждается, что всякий интуитивный алгоритм может быть реализован с помощью некоторой машины Тьюринга [1]. Иными словами, машина Тьюринга — достаточно точная модель компьютера общего назначения, хотя она и обладает определёнными ограничениями.

Одним из возможных доказательств того, что язык программирования называется тьюринг-полным, является возможность эмулировать на нём машину Тьюринга, и тем самым реализовать любой алгоритм [2]. Большинство современных языков программирования тьюринг-полные [3, 4]. Это верно, как языков полностью, например, для функционального Lisp, так и для конкретных используемых в них механизмов, например, шаблонов в C++. Получается, что большинство языков программирования взаимозаменяемы, или в общем случае — изоморфны. Возникает закономерный вопрос, а в чём тогда причина существования большого количества языков программирования.

Существуют различные подходы к написанию программ, называемые парадигмами. *Парадигма* — это метод, подход к формулировке задач (проблем) и путей их решения. *Парадигма в программировании* — это способ концептуализации, определяющий организацию вычислений и структурирование работы, выполняемой компьютером. Фактически это набор идей и правил, согласно которым следует разрабатывать программу. Следует отметить, что данные идеи могут пересекаться, поэтому различные парадигмы связаны между собой [5] и для корректного использования какой-либо из них стоит сначала изучить другие.

Сам термин в разрезе программирования впервые применил ещё в 1978 году Роберт Флойд в своей лекции лауреата премии Тьюринга [6]. На сегодняшний день создано большое количество языков, на которых можно писать в различных парадигмах. По данным индекса TIOBE насчитывается более 250 языков [7]. Столь широкое разнообразие можно объяснить несколькими причинами, в том числе:

- Необходимость работы программ на различных устройствах, со своими техническими особенностями.
- Удобство использования того или иного языка для решения какого-либо класса задач или, если говорить более обобщённо, написание кода с использованием различных парадигм.
- Эволюция ЭВМ и, как следствие, эволюция исполняемых команд и синтаксиса текста программ. Появление новых парадигм ведёт к созданию новых языков.

В основном современные языки программирования позволяют писать код в различных парадигмах, являются мультипарадигменными, но предполагают использование лишь тех из них, под которые тот или иной язык разрабатывался. Например, на языке С можно писать код в ООП или функциональной парадигмах, но это будет требовать дополнительных усилий и накладных расходов, поэтому, зачастую, лучше использовать процедурный подход, на который ориентирован данный язык.

В данной работе ищется взаимосвязь между различными языками программирования и предлагается способ конвертации кода между ними.

1. Материалы и методы

1.1. Изоморфизм алгоритмов

Сначала необходимо дать несколько определений.

Эквивалентность алгоритмов — свойство алгоритмов при каждом исходном данном, к которому применим хотя бы один из них, приводить к получению одинаковых результатов.

Выделяют сильную и слабую эквивалентность алгоритмов. Два алгоритма можно считать *слабо эквивалентными*, если у них одинакова область определения и различные систем правил при совпадении реализуемых ими функций. Если системы правил у двух алгоритмов также эквивалентны (не равны) при выполнении вышеописанных условий, то говорят о *сильной эквивалентности* алгоритмов.

Изоморфизм — взаимно однозначное соответствие между двумя множествами каких-либо объектов. Назовём два алгоритма изоморфными, если они решают одну и ту же задачу, но могут иметь различные формы, структуры или представления, иными словами — они являются слабо эквивалентными. В контексте изоморфизма алгоритмов «форма» означает, как алгоритм организован: какие структуры данных используются, какие операции выполняются и в каком порядке. «Представление» относится к тому, как код алгоритма записан на определенном языке программирования или как он представлен в виде блок-схемы, псевдокода и др., например:

1. *Сортировка пузырьком и сортировка вставками*. Оба алгоритма решают задачу сортировки, но используют разные методы для этого. Сортировка пузырьком сравнивает и обменивает соседние элементы, пока массив не будет отсортирован. Сортировка вставками поочередно вставляет каждый элемент в правильную позицию в отсортированной части массива.
2. *Бинарный поиск и интерполяционный поиск*. Оба алгоритма решают задачу поиска элемента в упорядоченном массиве. Бинарный поиск делит массив пополам и сравнивает искомый элемент с средним. Интерполяционный поиск использует интерполяцию для примерного определения положения элемента в массиве.

Изоморфизм алгоритмов подчеркивает, что существует несколько способов решения одной и той же задачи, и каждый из этих способов может быть оптимальным в разных ситуациях и при разных наборах требований и условий.

1.2. Изоморфизм языков программирования

Как было отмечено выше современный языки программирования являются мультипарадигменными, но каждый из них ориентирован всё равно лишь на несколько из них.

Рассмотрим некоторые парадигмы программирования и попробуем выделить их основные отличия и классы задач, в которых каждая из них наиболее актуальна для применения:

1. *Процедурное программирование*:

В данной парадигме программа представляет собой последовательность команд, которые можно собрать в более крупные блоки кода — процедуры.

2. *Объектно-ориентированное программирование (ООП)*:

Программа строится вокруг объектов, которые представляют реальные или абстрактные сущности, имеющие свойства и методы.

3. *Аспектно-ориентированное программирование (АОП)*:

Основная идея — разделить программу на модули, которые представляют собой срезы функциональности, называемые «аспектами». Аспекты могут перехватывать и изменять поведение других модулей.

4. *Функциональное программирование*:

Акцент на функциях как на основных строительных блоках программы. Функции рассматриваются как «первоклассные» объекты, т.е. они могут быть переданы как аргументы, возвращены как результат и др.

5. *Логическое программирование*:

Программа представляет собой набор логических утверждений, и исполнитель пытается найти значения, которые удовлетворяют этим утверждениям.

6. *Реактивное программирование*:

Акцент на потоках данных и распространении изменений. Программа реагирует на изменения в данных и автоматически обновляет интерфейс или другие компоненты.

Первые 3 приведённые выше парадигмы объединяются в *императивную парадигму*, где программа представляет собой последовательность команд, которые изменяют состояние программы. Для практических примеров алгоритмов в императивной парадигме в данной работе будет использоваться язык высокого уровня Python [8].

В разных парадигмах удобнее реализовывать те или иные алгоритмы с точки зрения компактности и читаемости кода, а, зачастую, и скорости его работы.

Задачи, связанные с поиском решений и логическим выводом, более компактными и естественными представляются в логической парадигме. В качестве примеров языков, её реализующих можно выделить Prolog-D [9, 10], который по сегодняшний день используется в том числе и в учебных целях и коммерческий swi-prolog [11].

Например, решение задачи о восьми ферзях [12] в логической парадигме с использованием Prolog представляется весьма естественно и компактно, так как она сосредотачивается на поиске корректной расстановки ферзей с учетом всех логических ограничений. Ниже в листинге 1 приведён пример реализации на языке swi-prolog.

```
:— use_module(library(clpfd)).
queens(N, Qs) :-
 length(Qs, N),
 Qs ins 1..N,
 safe(Qs),
 label(Qs).
safe([]).
safe([Q|Qs]) :-
 safe(Qs, Q, 1),
 safe(Qs).
safe([], _, _).
safe([Q|Qs], Q0, D0) :-
 Q0 #\= Q,
 abs(Q0 — Q) #\= D0,
 D1 #= D0 + 1,
 safe(Qs, Q0, D1).
?— queens(8, Qs).
```

Листинг 1. Решение задачи о расстановке 8 ферзей в логической парадигме на языке swi-prolog

В процедурной же парадигме точно такая же задача выглядит объёмнее и сложнее. Далее в листинге 2 приведён пример реализации на языке python:

```
def is_safe(board, row, col):
 # Проверяем вертикали
 for i in range(row):
 if board[i][col] == 1:
 return False
 # Проверяем левую диагональ вверх
 for i, j in zip(range(row, -1, -1), range(col, -1, -1)):
 if board[i][j] == 1:
 return False
 # Проверяем правую диагональ вверх
 for i, j in zip(range(row, -1, -1), range(col, len(board))):
 if board[i][j] == 1:
 return False
 return True
def solve_n_queens_util(board, row):
 n = len(board)
 if row >= n:
 return True
 for i in range(n):
 if is_safe(board, row, i):
 board[row][i] = 1
 if solve_n_queens_util(board, row + 1):
 return True
 board[row][i] = 0
 return False
def solve_n_queens(n):
 board = [[0 for _ in range(n)] for _ in range(n)]
 if not solve_n_queens_util(board, 0):
 return []
 return board
n = 8
solution = solve_n_queens(n)
if solution:
 print(solution)
else:
 print(«Решение не найдено»)
```

Листинг 2. Решение задачи о расстановке 8 ферзей в процедурной парадигме на языке python

Как видно из примеров выше в процедурной парадигме решение выглядит гораздо более объемным, так как требует явных циклов и условий для проверки различных условий. Таким образом, в примере с решением задачи о восьми ферзях логическая парадигма предоставляет более интуитивное и наглядное решение. В свою очередь в процедурной парадигме задачи, которые могут быть реализованы более компактно — это реализации алгоритмов, требующих множественные изменения состояния. В функциональной и логической парадигмах приходится использовать рекурсию, что может увеличить скорость вычислений и уменьшить наглядность. Ниже в листинге 3 представлен пример сортировки пузырьком в процедурной парадигме на Python.

```
def bubble_sort(arr):
 n = len(arr)
 for i in range(n):
 for j in range(0, n-i-1):
 if arr[j] > arr[j+1]:
```

```
    arr[j], arr[j+1] = arr[j+1], arr[j]
    return arr
```

Листинг 3. Сортировка пузырьком в процедурной парадигме на языке python

А в листинге 4 пример той же сортировки на языке swi-prolog.

```
sort_b([],[]) :— !.
sort_b([A], [A]) :— !.
sort_b(Arr, Result) :-
        bubbleToEnd(Arr, Arr), Result = Arr ;
        bubbleToEnd(Arr, Tmp), sort_b(Tmp, Result).
bubbleToEnd([], []) :— !.
bubbleToEnd([A], [A]) :— !.
bubbleToEnd([A,B|Terms], [B|TempRes]) :-
        B < A, bubbleToEnd([A|Terms], TempRes).
bubbleToEnd([A,B|Terms], [A|TempRes]) :-
        A =< B, bubbleToEnd([B|Terms], TempRes).
```

Листинг 4. Сортировка пузырьком в логической парадигме на языке swi-prolog

В сравнении с функциональной и логической парадигмами, где используются рекурсивные вызовы и более сложные конструкции, процедурная парадигма может предложить более прямолинейное и очевидное решение для алгоритмов, требующих множественные изменения состояния.

Если рассматривать функциональную парадигму, то в ней наилучшем образом выглядят задачи, связанные с манипуляциями над данными и обработкой коллекций. Пример подобной задачи — подсчет суммы всех элементов в списке. В функциональной парадигме можно использовать рекурсию и высокоуровневые функции, такие как map, filter, и reduce для выполнения этой задачи очень компактно. В листинге 5 приведён пример на языке Haskell.

```
sumList :: [Int] -> Int
sumList = foldr (+) 0
```

Листинг 5. Подсчет суммы всех элементов в списке в функциональной парадигме на языке haskel

Функция sumList принимает список целых чисел и возвращает их сумму. В данном примере используется функция foldr, которая применяет указанную бинарную операцию (в данном случае, сложение) к элементам списка, начиная с правого конца, с использованием начального значения (в данном случае, 0). В сравнении с императивной парадигмой, где нужно было бы использовать цикл и переменную для накопления суммы, функциональный подход гораздо более компактен. Таким образом, многие задачи обработки данных и манипуляции с коллекциями могут быть выражены более кратко и элегантно в функциональной парадигме благодаря высокоуровневым функциям и рекурсии.

ООП (Объектно-Ориентированное Программирование) позволяет абстрагировать данные и операции над ними в виде объектов, что может быть особенно удобным для определенных задач. Одной из таких задач является моделирование реальных сущностей с их характеристиками и методами.

В листинге 6 приведён пример реализации очереди с использованием ООП на Python.

```
class Queue:
 def __init__(self):
 self.items = []
 def enqueue(self, item):
 self.items.append(item)
 def dequeue(self):
 if self.is_empty():
 return None
 return self.items.pop(0)
 def is_empty(self):
 return self.items == []
 def peek(self):
 return self.items[0] if not self.is_empty() else None
```

Листинг 6. Реализация класса очередь в объектно-ориентированной парадигме на языке python

В этом примере, Queue — это класс, представляющий структуру данных «очередь». Он содержит методы enqueue (добавление элемента в конец очереди), dequeue (удаление элемента из начала очереди), is_empty (проверка на пустоту) и peek (получение элемента из начала очереди без удаления). В ООП парадигме, возможно создать объект типа Queue и использовать его методы для управления данными. Этот подход удобен и эффективен для организации кода, работающего с абстрактными структурами данных. В сравнении с другими парадигмами, в ООП мы можем выразить такую абстракцию более наглядно и интуитивно понятно.

Аспектно-Ориентированное Программирование (АОП) является парадигмой, которая позволяет разделять аспекты (или сквозные функциональные возможности) от основной логики программы. Одной из типичных задач, решаемых в АОП, является логирование. Пример логирования в Python с использованием библиотеки aspectlib приведён в листинге 7.

```
import aspectlib
@aspectlib.Aspect(bind=True)
def log_calls(f, *args, **kwargs):
 print(f'Calling {f.__name__} with args {args}, {kwargs}')
 yield
@log_calls
def foo(a):
 print("Inside foo with " + a)
```

Листинг 7. Пример логирования запросов в аспектно-ориентированной парадигме на языке python

В этом примере, log_calls — это аспект, который добавляет логирование к функции перед вызовом оригинальной функции. При его применении все вызовы foo будут логироваться. Аспектно-Ориентированное Программирование позволяет выражать такие сквозные функциональности (например, логирование, аудит, транзакции) более компактно и модульно, чем в других парадигмах.

Реактивное программирование (РП) ориентировано на работу с потоками данных и событиями. Одним из примеров, который может быть выражен более компактно в реактивной парадигме, является реактивный поиск. Пример в рамках RxJS (библиотеки для реактивного программирования в JavaScript) — это поиск элемента в потоке данных представлен в листинге 8.

```
const { from, of } = require('rxjs');
const { filter } = require('rxjs/operators');
const source = from([1, 2, 3, 4, 5]);
const searchValue = 3;
source.pipe(
 filter(value => value === searchValue)
).subscribe(
 result => console.log(`Найден элемент: ${result}`),
 error => console.error(error)
);
```

Листинг 8. Поиск элемента в массиве в реактивной парадигме на языке JavaScript с помощью библиотеки RxJS

В приведенном примере импортируются операторы RxJS и создаётся поток данных из массива. Затем с помощью оператора filter, выбираются только те значения, которые равны искомому элементу 3. Таким образом, был создан реактивный поток данных, с фильтрацией его по определенному условию и подпиской на результаты. В реактивной парадигме подобные задачи манипуляции с потоками данных и реакциями на события решаются более компактно и выразительно. Вместо использования императивных циклов и условий, происходит описание, какие операции необходимо применить к исходному потоку.

1.3. Методы перевода кода из одного языка в другой

Существует 2 основных способа провести перевод кода из одного языка в другой:

1. Ручной перевод: Разработчик может вручную переписать код на другом языке, следуя логике и структуре исходного кода. Это может быть долгим и трудоемким процессом и требовать хорошего понимания обоих языков, но дает полный контроль над переводом.
2. Использование инструментов автоматического преобразования: Некоторые инструменты позволяют автоматически преобразовывать код из одного языка программирования в другой. Однако они обычно не идеальны и требуют дополнительной правки и корректировки кода после преобразования.

При переводе кода важно уделять внимание тестированию, чтобы убедиться, что он работает корректно на новом языке. Также необходимо учесть различия в семантике и особенностях языков, чтобы избежать потенциальных ошибок и проблем в будущем.

Рассмотрим подробнее автоматической вариант миграции кода с одного языка программирования на другой. Автоматический переводчик кода — подход, который обеспечивает более быстрое и меньше подверженное человеческим ошибкам решение для перевода. Ниже представлены основные подходы к автоматическому переводу:

1. *Синтаксический анализ и генерация*: Этот подход основан на синтаксическом анализе и генерации кода на целевом языке на основе абстрактного синтаксического дерева (AST). Сначала производится анализ синтаксиса исходного кода и строится AST, а затем генератор создает эквивалентный код на целевом языке, исходя из полученной структуры данных.

Преимущества:

1. *Высокая точность*. Данный подход обычно обеспечивает высокую точность перевода, поскольку он базируется на анализе структуры исходного кода и создании эквивалентной структуры на целевом языке. Это помогает избежать большинства синтаксических ошибок.
2. *Поддержка сложных языковых особенностей*. Поскольку синтаксический анализ учитывает специфику каждого языка программирования, подобный подход способен обрабатывать сложные языковые конструкции и особенности, которые могут быть уникальными для каждого языка.
3. *Гарантированное соблюдение стандартов*. При правильной реализации этот подход может гарантировать соблюдение стандартов целевого языка программирования, что облегчает поддержку и дальнейшую разработку кода на новом языке.

Недостатки:

1. *Трудоемкость разработки*. Реализация синтаксического анализа и генерации кода для каждой пары исходного и целевого языков является трудоемкой задачей. Она требует значительных усилий в разработке и поддержке подобных инструментов.
2. *Сложность обновлений*. При изменениях в синтаксисе или структуре целевого языка програм-

мирования могут потребоваться значительные усилия для обновления инструмента перевода.

3. *Невозможность обработки семантических аспектов*. Этот подход фокусируется преимущественно на синтаксическом уровне, и ему сложно обрабатывать семантические аспекты кода. Ошибки, связанные с семантикой, могут остаться незамеченными.

В целом, подход, основанный на синтаксическом анализе и генерации кода, подходит для перевода между языками с схожей структурой и синтаксисом, но может быть сложен в случаях, когда языки сильно отличаются.

2. *Использование метасимволов и шаблонов*. Данный подход предполагает определение образцов (шаблонов) на исходном языке и их замену эквивалентными шаблонами на целевом языке, что можно делать с использованием регулярных выражений или других методов сопоставления текста.

Преимущества:

1. *Относительная простота разработки*. Создание и использование шаблонов и метасимволов является более простым способом автоматического перевода, чем разработка сложных синтаксических анализаторов и генераторов кода.
2. *Гибкость*. Данный подход позволяет создавать уникальные шаблоны и правила для перевода конкретных конструкций или паттернов между языками, т.е. имеется возможность адаптировать перевод под конкретные дополнительные требования.

Недостатки:

1. *Ограниченность по точности*. Подход, основанный на шаблонах, может быть менее точным, чем синтаксический анализ, особенно при переводе сложных и контекстозависимых конструкций.
2. *Сложность управления языковыми особенностями*. Перевод кода часто требует учета не только синтаксических, но и семантических аспектов, которые сложно учесть при подобном подходе.
3. *Зависимость от качества шаблонов*. Качество перевода будет зависеть от качества и полноты созданных шаблонов. Если набор шаблонов неполон или не обрабатывает все возможные случаи, это может привести к неправильным переводам.
4. *Сложность обработки изменений в языках*. При изменениях в синтаксисе или семантике языков перевод на основе шаблонов также потребует обновлений и адаптаций.

В целом, подход, основанный на метасимволах и шаблонах, хорошо подходит для относительно простых и структурированных языков и может быть полезным для быстрого прототипирования переводчика. Однако он может ограничивать точность и сложность перевода в более сложных случаях.

3. *Использование трансляторов и компиляторов*. Некоторые языки программирования предоставляют инструменты, такие как трансляторы или компиляторы, которые способны переводить код с одного языка на другой. Например, в случае перевода кода с языка C на C++, можно использовать компилятор C++ с определенными флагами для обеспечения совместимости с C.

Преимущества:

1. *Высокая точность и надежность*. Трансляторы и компиляторы обычно обеспечивают высокую точность перевода, поскольку они разработаны и поддерживаются профессиональными командами с учетом стандартов языков, что минимизирует вероятность появления ошибок в переводе.
2. *Соответствие стандартам*. Такие инструменты стремятся соблюдать стандарты языков, что облегчает соблюдение правил и норм кодирования в целевом языке.
3. *Высокая производительность*. Трансляторы и компиляторы часто оптимизируют генерацию кода, что может привести к более производительному переводу.
4. *Поддержка множества языков*. Некоторые трансляторы и компиляторы спроектированы для поддержки множества языков, что делает их гибкими и мощными инструментами для перевода.
5. *Автоматическое обновление*. Если целевой язык обновляется, разработчики трансляторов и компиляторов обычно выпускают обновления, чтобы поддерживать совместимость.

Недостатки:

1. *Ограниченность доступными инструментами*. Не для всех пар языков программирования существуют готовые трансляторы или компиляторы, поддерживающие именно требуемые языки.
2. *Сложность разработки и поддержки*. Создание и поддержка транслятора или компилятора — это сложная и ресурсоемкая задача. Она требует опыта и знаний в области компиляторных технологий.
3. *Не всегда подходит для перевода сложных семантических аспектов*. Трансляторы и компиляторы могут быть ограничены в способности перевода сложных семантических конструкций между языками, поскольку они обычно фокусируются на синтаксисе.
4. *Требует перекомпиляции*. При использовании данного подхода, переведенный код обычно требует перекомпиляции, что может вызвать неудобства в некоторых сценариях.

В целом, использование трансляторов и компиляторов может быть очень эффективным для перевода кода между языками, но данный подход имеет свои ограничения и требует более высокого уровня экспертизы и ресурсов для разработки и поддержки.

4. *Автоматический переводчик с использованием машинного обучения*. Подобный подход включает в себя применение методов машинного обучения, таких как нейронные сети для автоматического перевода кода. Модель обучается на парах разных языков и структурирует свои прогнозы на основе данного обучения.

Преимущества:

1. *Адаптивность и обучаемость*. Модели машинного обучения способны «учиться» на основе данных. Это означает, что они могут адаптироваться к различным парям языков программирования и улучшать свою производительность с течением времени, поскольку им предоставляются новые данные для обучения.
2. *Гибкость и обработка сложных случаев*. Модели машинного обучения могут обрабатывать сложные синтаксические и семантические конструкции, которые могут быть трудными для обработки другими методами, такими как шаблоны или синтаксический анализ.
3. *Автоматическое обнаружение паттернов*. Модели машинного обучения могут автоматически обнаруживать паттерны и связи между языками, что делает их способными к обработке сложных языковых конструкций.

Недостатки:

1. *Требуется большое количество данных*. Для обучения моделей машинного обучения требуется большое количество данных, размеченных на парах различных языков. Если данных недостаточно, это может привести к низкой производительности и неточным переводам.
2. *Сложность в интерпретации*. Модели машинного обучения могут быть сложными и часто рассматриваются как «черные ящики», что делает затруднительным понимание и интерпретацию процесса перевода.
3. *Необходимость перепроверки результатов*. Нейронные сети позволяют найти только субъоптимальное решение, и соответственно неприменимы для задач, в которых требуется высокая точность. Иными словами, нет никакой гарантии, что полученный сконвертированный код будет работать, как ожидается.
4. *Не всегда могут соблюдаться стандарты языка*. Модели машинного обучения могут генерировать код, который не всегда соответствует стандартам языка программирования, что может потребовать последующей ручной доработки.
5. *Результат не является постоянным*. В зависимости от архитектуры нейронной сети, каждая из попыток конвертации кода, может выдавать различные результаты. Данная особенность может не позволить использовать подобный инструмент, например, в образовательных целях.

В целом, использование машинного обучения для автоматического перевода кода обладает большим потенциалом для улучшения точности и гибкости перевода, но требует больших объемов данных и обязательной последующей проверки получившегося кода.

5. *Использование промежуточного представления (Intermediate Representation)*. В некоторых случаях особенно при переводе кода между языками схожей архитектуры, можно использовать промежуточное представление, которое абстрагирует детали конкретного языка. Это представление затем может быть переведено на целевой язык.

Преимущества:

1. *Сохранение структуры и семантики*. Промежуточное представление (IR) обычно представляет собой абстракцию, которая может соблюдать структуру и семантику исходного кода. Это обеспечивает более точный перевод с сохранением семантических аспектов.
2. *Универсальность*: Промежуточное представление может использоваться для перевода между множеством языков программирования, что делает его универсальным инструментом для многих задач перевода.
3. *Обработка сложных языковых особенностей*. Поскольку IR обычно более абстрактно и меньше зависит от конкретных языковых особенностей, он может легче обрабатывать сложные синтаксические и семантические конструкции.

Недостатки:

1. *Сложность разработки и поддержки IR*. Создание и поддержка промежуточного представления может быть сложной задачей, требующей значительных усилий и экспертизы в области компиляторных технологий.
2. *Дополнительные этапы обработки*. Использование IR добавляет дополнительный этап обработки в процессе перевода, что может замедлить процесс.
3. *Зависимость от структуры IR*. Точность и эффективность перевода будут зависеть от того, насколько хорошо промежуточное представление отображает структуру исходного кода.
4. *Ограниченность языков программирования*. Не всегда существует подходящее промежуточное представление для всех пар языков програм-

мирования, особенно для менее популярных или экзотических языков.

5. *Требует усилий по обновлению и сопровождению*. Как и другие подходы, использование IR требует постоянного обновления и сопровождения, особенно при изменениях в языках программирования.

Пятый подход с использованием промежуточного представления может быть эффективным при переводе между языками с разной синтаксической и семантической структурой, но он требует более глубокого понимания компиляторных технологий и создания соответствующего промежуточного представления для конкретных задач перевода.

2. Результаты

Следует подчеркнуть, что ранее рассмотрены далеко не все существующие парадигмы и подходы к переводу кода из одного языка в другой, но представленных примеров достаточно, чтобы сделать вывод, что реализация одного и того же алгоритма изоморфна в различных парадигмах и, как следствие, языках. Иными словами, изоморфизм алгоритмов переходит в изоморфизм языков программирования, а значит, существует возможность выделения общего, глобального языка, который позволит переводить код из одного языка в другой, иными словами, с его помощью можно будет трансформировать реализацию алгоритмов между различными парадигмами. Ниже приведён список ситуаций, когда данный перевод может принести пользу:

— появится возможность на практике без больших трудозатрат улучшить характеристики уже написанных программ. Например, увеличить скорость их выполнения, уменьшить объём необходимой памяти или исходного кода;
— в учебных целях имеет смысл сравнивать как парадигмы написания программ, так и сами языки программирования;
— появится возможность унифицировать запуск программ на различных машинах без использования каких-либо дополнительных инструментов (например, виртуальных машин). Достаточно будет перевести программу на требуемый какой-либо платформой язык.

Также можно сделать вывод, что использование глобального языка для трансформации кода из одного языка программирования в другой обеспечит высокую точность перевода без необходимости дополнительной перепроверки результатов, и данный подход применим для любых языков. Отдельно следует отметить, что при использовании глобального языка нет необходимости создавать инструменты для перевода каждого языка в каждый. Достаточно научиться переводить языки в глобальный и наоборот. Например, для 250 языков получится 250 трансформаций вместо 31125.

Заключение

В качестве глобального можно выбрать один из уже существующих языков, однако, как было показано выше, они все создавались и развиваются для решения каких-либо классов задач. Соответственно в них используются и появляются новые возможности, которые нельзя напрямую перевести в другой язык. Например, нельзя простыми заменами синтаксиса 1 к 1 перевести программу из языка prolog в Си. Для этого потребуется дополнительный анализ и трансформации, которые переведут код из логической парадигмы в процедурную. Как следствие — выбор в качестве глобального уже существующего языка может быть неудобным с точки зрения необходимости подстраивать перевод под определённую парадигму или его особенности.

Таким образом лучшим решением является создать новый язык, который будет развиваться именно для задачи перевода кода из одного языка программирования в другой. Иными словами, он будет создаваться специально из расчёта, чтобы было наиболее удобно выполнять унификацию кода, написанного на различных языках и согласно разнообразным парадигмам.

Выполнять подобный перевод можно на различных уровнях: начиная от синтаксиса и заканчивая переводом машинных инструкций. Наиболее эффективным и доступным способом выбран перевод именно синтаксиса, так как он:

— не зависит от компьютера, на котором запускается код и от технологий, требуемых для запуска, в том числе jvm;
— не зависит от того какой язык выбран в качестве исходного: компилируемый или интерпретируемый;
— позволяет переводить логику программы на самом высоком уровне кода, что гораздо нагляднее, чем работа с уже обработанными компилятором или транслятором командами. Данная особенность позволит привлечь к разработке глобального языка большее число людей и сделает его развитие более быстрым и наглядным.

Отдельно следует обратить внимание, что создание глобального языка позволит обрести хорошую точку входа для обучения программированию.

ЛИТЕРАТУРА

1. Марголин И.Д., Дубовская Н.П. Основные этапы развития искусственного интеллекта / Марголин И.Д. // Молодой Учёный. — Казань, 2018. — Вып. 20. — С. 23–25.
2. Sherron W. Methods for Demonstrating Turing Completeness of a Programming Language [Electronic resource], URL: https://copyprogramming.com/howto/how-to-prove-a-programming-language-is-turing-complete (дата обращения 18.10.2023)
3. Banzhaf W., Nordin P., Keller R.E., Francone F.D. Genetic Programming: An Introduction on the Automatic Evolution of computer programs and its Applications — Morgan Kaufmann, San Francisco, 1998.
4. Turing Completeness, [Electronic resource], URL: https://www.cs.odu.edu/~zeil/cs390/latest/Public/turing-complete/index.html (дата обращения 18.10.2023)
5. Greg M. Programming Paradigms, Turing Completeness and Computational Thinking — arXiv, 2020.
6. Floyd W. Turing Award Lecture, 1978: The Paradigms of Programming — Communications of the ACM, Vol.22:8, 1979. 455-460 p.
7. TIOBE Programming Community Index Definition. [Electronic resource], URL: https://www.tiobe.com/tiobe-index/programminglanguages_definition/ (дата обращения 18.10.2023)
8. Фёдоров Д.Ю. Программирование на языке высокого уровня Python. — М.: Издательство Юрайт, 2022.
9. Григорьев С.Г., Уртминцев А.Г. Язык программирования Пролог — Д.// Каталог ВДНХ СССР — Москва, 1988.
10. Григорьев С.Г., Уртминцев А.Г. Пролог — Д и учебные экспертные системы// в сб. У Всесоюз. Семинар «ПЭВМ в учебном процессе», — М. ИПИАН СССР — 1989.
11. Swi-prolog [Electronic resource], URL: https://www.swi-prolog.org/ (дата обращения 18.10.2023)
12. Jordan B., Brett S. A survey of known results and research areas for n-queens — Discrete Mathematics, Vol. 309. 1–31 pp.

Журнал «Современная наука: актуальные проблемы теории и практики»