NEXTGEN: ЯЗЫК С ПРОГРАММИРУЕМОЙ СЕМАНТИКОЙ**А.Д. Городецкий***Омский государственный университет им. Ф.М. Достоевского, Омск, Российская Федерация*

Рассмотрены подходы к заданию семантики в языках программирования. Предложен альтернативный вариант в виде программируемой подсистемы анализа потока данных, позволяющей задавать семантику, ограничивающую нужные классы ошибок.

Ключевые слова: семантика языка программирования, системы типов, анализ потока данных.

NEXTGEN: PROGRAMMING LANGUAGE WITH PROGRAMMABLE SEMANTICS**A.D. Gorodetskiy***Dostoevsky Omsk State University, Omsk, Russian Federation*

Approaches to defining semantics in programming languages are considered. An alternative option is proposed in the form of a programmable data flow analysis subsystem, which allows setting semantics that limit the necessary error classes.

Keywords: semantics of a programming language, type systems, data flow analysis.

Создатели языков программирования уже более 50 лет решают одну и ту же задачу – уменьшение количества ошибок в программах времени исполнения. Полностью эта задача считается неразрешимой в силу полноты по Тьюрингу, однако современные языки программирования еще на этапе компиляции могут распознавать много классов ошибок.

Первый и самый главный класс ошибок – это неверная интерпретация данных. Первые языки высокого уровня ввели понятие системы типов, ограничивающее произвольную интерпретацию данных, что дало огромное преимущество над ассемблерами. Архитектура существующих ЭВМ создает следующие задачи корректности, которые хотелось бы решать на этапе компиляции:

- управление динамической памятью;
- управление ресурсами – файловые дескрипторы, захват устройств, мьютексов и вообще каких-либо ресурсов, за которые есть конкуренция между процессами;
- гонки данных;
- оборачивание аппаратных высокопроизводительных API в безопасный интерфейс.

Список, указанный выше, неполон и растет по мере развития индустрии. Множество системных API также открытое.

На данный момент мейнстримовые языки решают эти задачи в одном направлении – выстраивая некоторую аксиоматику на уровне системы типов. На ее основе можно синтезировать некоторую семантику, позволяющую определять некоторый класс «некорректных» программ [1].

Чистые функциональные языки программирования, например, на корню решают проблемы с гонками данных, запрещая их изменяемость, однако сильно сужают множество возможных программ, что доставляет сложности в реализации большого числа алгоритмов.

Примечательно, что язык Rust строит свою аксиоматику не вокруг изменяемости данных, а вокруг концепции владения данными и времени жизни данных. Такая аксиоматика точнее отражает инварианты аппаратного обеспечения.

Третий подход – когда среда исполнения языка программирования дает возможность поддержки нужных инвариантов, но ценой потери производительности.

Указанные выше подходы частично решают поставленные задачи, но главная проблема состоит в том, что они расширяются только на уровне спецификации языка, что является процессом крайне сложным и часто невозможным [1–3].

Рассмотрим совершенно другой подход:

1. У нас есть некоторый язык, аксиоматика которого решает *только* вопросы с корректностью интерпретации данных.

2. Компилятор этого языка производит анализ потока данных (DFA), и позволяет разработчикам библиотек устанавливать некоторые инварианты на переменных, что позволяет «программировать» добавочную семантику языка.

Рассмотрим классическую проблему работы с закрытыми файлами:

```
# library code
def fopen = path: String, mode: String do
  # implementation
  DFA.nocopy(file,
    'file descriptor cannot be copied to another var')

def fclose = file: File do
  # implementation
  DFA.die(file, 'file was closed')

# user code
def main =
  file = fopen('/tmp/xxx', 'r')
  readSome(file)
  fclose(file)
  readSome(file)
  #^^^ DFA compilation error: cannot use dead variable
  'file' because of 'file was closed'
```

Данная схема обладает следующими преимуществами:

1. Правила DFA только уменьшают количество корректных программ, а не увеличивают.

2. Позволяет выразить практически любые ограничения на отношениях переменных, что дает разработчиками библиотек полную свободу в дизайне эффективных и корректных API.

3. Расширение правил DFA не расширяет язык в привычном смысле, то есть не изменяет грамматику и общую семантику.

4. Применение правил DFA локально, то есть их применение на одних переменных никак не влияет на другие, в отличие от изменения аксиоматики систем типов.

СПИСОК ЛИТЕРАТУРЫ

1. Компиляторы: принципы, технологии и инструменты // А. Ахо, Р. Сети, Дж. Ульман. М.: Вильямс, 2001. 768 с.
2. Типы в языках программирования / Б. Пирс. М.: Лямбда, 2011. 680 с.
3. Advanced Compiler Design And Implementation // Steven S. Muchnik. Morgan Kaufmann Publishers, 1997. 880 p.