YAL Programming Language

Оглавление

Введение

Функциональные языки программирования набирают все большую популярность в профессиональной среде. К примеру, небезызвестная компания Facebook (ныне Meta) использует функциональный язык программирования Haskell для борьбы со спамом и с иными видами “junk information” в своих продуктах.

Функциональные языки программирования ценятся за стабильность, понятность, читаемость кода, низкий порог вхождения (в случае достаточности математического аппарата программиста), элегантность и чистоту (если таковая характерна конкретному языку). В случае реализации программного обеспечения (как правило, кроме низкоуровневых вычислений и операций) функциональные языки позволяют программисту выразить свои идеи в наиболее абстрактном виде.

Именно поэтому я считаю, что функциональные языки программирования должны быть широко известны и используемы в тех отраслях, где это наиболее выгодно и необходимо. А для изучения принципов их работы и основных реализационных моментов нет лучше метода, чем самостоятельная разработка подобного языка.

Цели и задачи

Цель проекта – реализация чистого статически типизированного функционального языка программирования с системой вывода типов по Хиндли-Милнеру (см. следующий раздел).

Задачи:

1. Создание концепции и синтаксиса языка.
2. Создание модели выполнения (денотационной семантики).
3. Реализация текстового анализатора (парсера).
4. Реализация систем типизации и вывода типов.
5. Реализация интерпретатора.

Подзадачи:

1. Изучение основных понятий теории типов.
2. Изучение полезных принципов и приемов для реализации алгоритма вывода типов.

Раздел 1. Основополагающие понятия

1.1 Глоссарий[1]

1. Функциональное программирование – это программирование, непрерывно связанное с математическими функциями и математикой в целом (один из вариантов определения).
2. Объекты первого порядка – объекты, которые могут быть переданы функции в качестве аргумента.
3. Чистота языка программирования[2] – отсутствие побочных эффектов.
4. Побочный эффект – любые действия компилятора\интерпретатора, изменяющие среду выполнения. Например: получение доступа и изменение состояния процесса, изменение значения переменной.
5. Статическая типизация – приём, при котором функция, переменная и любое другое значение связывается с соответствующим ему типом в момент объявления, и тип значения не может быть изменен позднее.
6. Тип – некоторое множество значений и соответствующих им операций (функций).
7. Вывод типов – алгоритм типизации терма, как правило, основывающийся на тесной взаимосвязи синтаксиса языка и типизируемых конструкций.
8. Терм – выражение, конструкция языка.
9. Семантика – некоторая модель выполнения программы, которая может придать терму определенный смысл (значение). Семантика подразделяется на два вида: операционная и денотационная.
10. Операционная семантика – определяет конечное состояние абстрактной машины, к которому она придет после некоторого прохода по терму в начальном состоянии.
11. Денотационная семантика – в качестве терма, как правило, выступает некий математический объект (например, число или функция) в некоторой уже существующей семантической области, а

---

[1] Я подразумеваю, что встретив какой-либо непонятный термин в тексте, вы вернетесь к глоссарию и узнаете больше о его смысле, значении. Также подразумевается, что большинство понятий основываются на данных того или иного ресурса, указанного в списке литературы.

[2] В нашем случае это почти соблюдается (см. примечание 1.1).

функция интерпретации, в свою очередь, соотносит эти термы с их эквивалентами в целевой области. Таким образом, она как бы указывает, денотирует (или представляет) терм в этой эквивалентной области.

12. Парсер (текстовый анализатор) – алгоритм (или совокупность алгоритмов), превращающий данный текст программы в соответствующее ему абстрактное синтаксическое дерево.

13. Абстрактное синтаксическое дерево – некоторое дерево, в котором (как правило) вершины сопоставлены с некоторыми операциями, а листья являются их аргументами, значениями, выражениями.

14. Дерево (в информатике) – широко используемая структура данных, имитирующая древовидную структуру в виде связных узлов.

15. Наивная теория множеств – раздел математики, изучающий общие свойства множеств.

16. Мемоизация – сохранение результатов вычислений функций с целью предотвратить их повторное выполнение.

17. Сборка мусора (в информатике) – процесс автоматизированного управления с памятью, аллокация полезных и деаллокация неиспользуемых ресурсов.

18. Каррирование – термин в информатике, получивший название в честь американского математика Хаскелла Карри, обозначающий сведение функции множественного аргумента к набору функций от одного аргумента, применение которых в правильном порядке дает тот же результат, что и изначальная функция.

19. Полиморфизм – возможность функции принимать и возвращать данные разных (общих) типов.

20. Лямбда-исчисление – формальная система вычислений, разработанная американским математиком Алонзо Черчем для исследования понятия вычислимости.

21. Грамматика языка (в информатике, условно) – формальный способ задания его синтаксических конструкций. То есть, возможность разделения множества всех встречаемых конструкций на два подмножества: валидных и невалидных конструкций.

1.2 Почему нам нужна статическая типизация

Язык программирования нуждается в наличии системы типов (подразумевается статической). “Почему это?” – спросит кто-то. Все довольно просто. К примеру, в слабо типизированных и нетипизированных языках программирования частой может стать проблема несоответствия желаемого действительному: например, умножив строку на число, мы получим далеко не численное значение. “Но, – воскликнут некоторые программисты, – ведь это же совершенно обычная практика!”. Если для кого-то это и вправду удобно и нормально, то не для всех и, точно, не всегда. С такой системой типизации нередким может стать и сложение числа с символом, так как в большинстве подобных языков также есть и неявное преобразование (implicit coercion; например язык программирования C++), которое выражается в преобразовании (если такое возможно) типа аргумента в другой тип, подходящий для данной ситуации. Таким образом, число останется числом, а вот символ внезапно станет целочисленным аргументом, который с большой радостью будет участвовать в дальнейших вычислениях и не выкинет никакого сообщения об ошибке на стек (разве, может быть, предупреждение). Также система типизации предотвращает разнообразные “опечатки”: например, использование номера вместо цены товара будет невозможным, если они имеют различные типы. Говоря о функциональных языках, хочется заметить, что функция является объектом первого порядка. То есть мы можем передать функцию в другую функцию как аргумент. Но что же тогда будет, если мы решим сложить функцию с числом? Конечно, в некоторых довольно развитых функциональных языках программирования, например в Haskell, существуют такие понятия, как классы типов, которые предназначены как раз таки для решения подобных проблем (см. LYH стр. 51). Но в нашем случае это просто недопустимо.

Исходя из этого, мы понимаем, что система типов, статическая система типов очень важна для реализации хорошего функционального языка программирования, так как заметно упрощает реализацию семантики языка (см.раздел 2.6) и избавляет нас от неожиданных результатов.

1.3 Краткие основы теории типов

Теория типов – широко развивающаяся наука, основывающаяся на наивной теории множеств. Главная её задача – классификация выражений с помощью определённых типов. Часто можно услышать термины “имеет тип”, “будет выведен” и прочее. Что же все это значит? Иметь тип, то есть отношение типизации (::), говорит нам о том, что некоторое значение, будто переменная, функция или что-то еще, имеет определенный тип, то есть связано с ним в контексте текущей среды типизации (typing environment)[3]. “Будет выведен” или же вывести тип – значит с помощью определенного алгоритма типизации, пройдя по терму, определить, какой тип он имеет в контексте текущей среды.

1.4 Алгоритм вывода типов по Хиндли-Милнеру

Робин Милнер (1934 - 2010)

Алгоритм в своем начальном представлении был предложен в 1958 году математиком Хаскеллом Карри, разработавшим его для типизированного лямбда-исчисления. Позднее, в 1969 году Роджер Хиндли расширил алгоритм и доказал, что он выводит наиболее общий тип выражения. В 1979 году Робин Милнер доказал эквивалентность свойств иного

[3] Если формальное определение отношения типизации непонятно на данном этапе, то оно будет ещё раз объяснено на конкретных примерах в следующем разделе.

алгоритма. И, наконец, в 1985 году Луис Дамас доказал, что алгоритм Милнера является законченным и правильно выводит полиморфные типы.

Основным наблюдением является тесная взаимосвязь типизируемых конструкций и их типов, что мы увидим позднее (см. раздел 2.5). Легкость реализации и использования, а главное, модификации заслуженно сделали данный алгоритм наиболее используемым в большинстве практических задач.

Раздел 2. Реализация языка программирования

2.1 Название языка

В соответствии с довольно популярной традицией среди программистов добавлять к названию проекта или продукта идиоматический префикс “Yet Another”, было решено назвать реализуемый язык “Yet Another Language”, дословно означающий “Еще Один Язык”. Впоследствии, будем придерживаться акронима “YAL”.

2.2 Выбор языка программирования

Для реализации поставленной цели был выбран функциональный язык программирования Haskell. Его основные плюсы заключаются в стабильности, чистоте, элегантности, ленивости и в наличии статической системы типизации. Также, в Haskell есть удобный инструмент под названием pattern matching (сопоставление с образцом), позволяющий создать ветвление функции в соответствии с входными данными, точнее, с их “форматом” (см. LYH стр. 60).

2.3 Грамматика языка

Грамматика языка по сути полностью определяет то, в какой форме нам предстоит писать код на нём. Поэтому было решено выбрать наиболее обычный вариант: ML-подобную грамматику с небольшими изменениями.

Запишем её основные моменты формально:

```
expr ::=
        literal                                 ; литерал
```

| var ; переменная
| con ; конструктор
| “lam” pat “→” expr ; лямбда-абстракция
| “\” pat “→” expr ; лямбда-абстракция
| “let” decl “in” expr ; let-выражение
| “if” expr “then” expr “else” expr ; if-выражение
| “case” exprs “of” (pats → expr)+[4] ; case-выражение
| expr op expr ; инфиксная запись
| expr expr ; применение выражений

exprs ::=
expr ; “свободное” выражение
| expr “,” exprs ; последовательность

pat ::=
varp ; паттерн переменной
| conp ; паттерн конструктора
| wildcard ; паттерн “wildcard”
| litp ; паттерн литерала

pats ::=
pat ; “свободный” паттерн
| pat pats ; последовательность
| ; отсутствие паттерна

Соответственно, pats можно было бы записать как pat+, то есть последовательность из нуля или нескольких элементов. А exprs – как expr*, то есть последовательность из одного или нескольких элементов.

[4] см. следующую страницу

```
decl ::=
        name pats+ “=” expr                ; декларация константы
      | name “::” type                     ; декларация типа
      | ext                                ; расширение

type ::=
        tvar                               ; типовая переменная
      | tcon                               ; типовая константа
      | type “→” type                      ; “стрелка” из типа в тип

program ::= decl*                          ; программа
```

2.4 Реализация текстового анализатора

Представив грамматику языка в формальном виде, мы смело можем реализовывать тестовый анализатор. Для его создания пришлось изучить полезную библиотеку Megaparsec (https://hackage.haskell.org/package/megaparsec, Mark Karpov), которая и была использована для реализации текстового анализатора (далее парсера). Выбор этой библиотеки был очевиден, так как она предоставляет удобный интерфейс, приемлемую скорость работы и хорошие сообщения об ошибках. Парсер получился довольно большим, по сути “копирующим” грамматику, реализующим некоторые совершенно технические моменты, поэтому его исходный код можно изучить по адресу: https://github.com/geothecode/YAL/blob/master/yal/source/Parsing.hs.

2.5 Реализация алгоритма вывода типов

Так как было решено реализовывать алгоритм вывода типов по Хиндли-Милнеру, то пришлось разобраться, как он работает. С этой целью я прочитал небольшую часть книги Бенджамина Пирса, но в большей мере мне понравились лекции Виталия Брагилевского (см. список литературы: 1-2 лекции). Основная часть алгоритма основывается на наблюдении, что

тип большинства конструкций как-либо взаимосвязан с самой конструкцией.

Например, рассмотрим конструкцию: if a then b else c. И пусть функция type(a) возвращает нам тип выражения.

1) По канонам функционального программирования if-конструкция имеет один определенный тип, то есть и then-ветка, и else-ветка, и само выражение имеют одинаковый тип. Поэтому заносим в среду “ограничение” (constraint – английский аналог), что type(b) равен type(c), а также, что все выражение – некоторая переменная на типах, которая впоследствии будет заменена на конечный тип.
2) Далее, понимаем, что перед нами условная конструкция, а значит где-то все-таки должно быть условие! В информатике придерживаются типа Bool (или Boolean) для логических выражений. Поэтому заносим в среду constraint, что type(a) равен Bool.
3) Вычисляем типы и решаем систему “ограничений”.
4) Получаем некоторую “подстановку на типах” и применяем ее к среде.
5) Таким образом, получаем конечный тип.

Формально, обдумывая данные шаги, можем указать всего три основных:

1) Вывод типов в выражении и сбор “ограничений”.
2) Решение системы уравнений (“ограничений”).
3) Применение подстановки.

Чтобы более наглядно показать взаимосвязь конструкций языка и их типов, рассмотрим еще несколько примеров.

1) x = 2
    a) Запись в среду: x :: a (x имеет тип a)
    b) Вывод: 2 :: Int
    c) Ограничение: a = Int
    d) Решение уравнений: a = Int
    e) Применение подстановки на среду: x :: Int
2) f p = 3

a) Запись в среду: “f :: a → b” (стрелка из a в b, то есть тип функции из a в b)
b) Вывод: p :: c, 3 :: Int
c) Ограничение: a = c, b = Int (возвращаемое значение всегда является последним из типов, в нашем случае это b)
d) Решение уравнений: a = c, b = Int
e) Применение подстановки: “f :: c → Int” или “f :: a → Int”

Как можно заметить, вывод типа конструкции довольно сильно зависит от самой конструкции, что и требовалось показать. Но тип f :: a → Int – это, кончено, тип, хотя на практике, когда тип полностью выведен, мы “закрепляем” его, чтобы никакая последующая подстановка не сделала a, например, Int. Делаем мы это с помощью введения схем (scheme), которые добавляют некий надуровень типа, говорящий нам о полиморфности определённых переменных. Выглядит это следующим образом:

“f :: ∀ a. a → Int”, хотя на практике такой символ неудобно писать, поэтому чаще это просто “f :: forall a. a → Int”.

Решение системы уравнений происходит с помощью алгоритма Робинсона для решения конечных эрбрановских термов, в нашем случае его частный случай для уравнений на типах (см. Виталий Брагилевский – лекция №1, 1:17:00). Подстановка же применяется тривиально.

Так как большинство технических моментов не нуждаются в объяснении из-за своей узкой специфики, то, полагаю, что основная идея реализации данного алгоритма понятна. Исходный код реализованного алгоритма находится по адресу:

https://github.com/geothecode/YAL/blob/master/yal/source/Typing.hs.

2.6 Реализация денотационной семантики

Правильная реализация денотационной семантики, да и любой семантики в целом – важная задача, к которой нужно приложить усилия. К счастью, часть возникающих проблем решает наличие статической типизации: нам не придется проверять типы “на месте” или при передаче в функцию, нам

не нужно реализовывать различных методов приведений типов (implicit или explicit coercion), хотя coercion – полезная вещь в некоторых ситуациях (https://repository.brynmawr.edu/cgi/viewcontent.cgi?article=1015&context=compsci_pubs). В нашем случае, помимо всего прочего, мы реализуем полезную функцию языка – сравнение с образцом (pattern matching), что также приходится учитывать при выполнении вычислений и выведении типа выражения.

Выполнение вычислений происходит приблизительно следующим образом:

1) В соответствии с форматом выражения выбираем правильную ветку
    a) Литерал ⇒ возвращаем.
    b) Переменная ⇒ ищем, чему она соответствует в глобальной среде выполнения, и возвращаем это значение.
    c) Лямбда-абстракция ⇒ возвращаем в новом “виде” с локальной средой.
    d) Применение ⇒ вычисляем “левую” и “правую” части и добавляем правую в локальную среду левой, если это возможно, или совершаем какое-то действие (вывод на консоль и тому подобное).
2) Возвращаем вычисленное значение (понятие значения довольно обширно и не заканчивается на одних только литералах, хотя это и довольно частый случай).

Сравнение с образцом происходит по довольно тривиальной модели, которую легко себе представить.

Исходный код модуля для вычисления значений: https://github.com/geothecode/YAL/blob/master/yal/source/Evaluation.hs.

Исходный код модуля для сопоставления с образцом:

https://github.com/geothecode/YAL/blob/master/yal/source/PatternMatching.hs.

2.7 Почему я не реализовал мемоизацию и ленивые вычисления

Ленивые вычисления – это одна из интересных функциональных особенностей языка Haskell, позволяющая работать с бесконечными или очень большими последовательностями данных, оптимизировать код.

Мемоизация – полезный подход оптимизации вычислений, основывающийся на сохранении вычисленных значений функций.

Так почему же я не стал реализовывать эти полезные инструменты? При реализации ленивых вычислений и, соответственно, мемоизации возникает проблема, справиться с которой довольно тяжело. Проблема заключается в борьбе с мусором, а именно с определением того, какие данные нужно оставить в куче, а что из нее надо удалить. Для борьбы с этим в 1959 году был придуман и использован сборщик мусора. Но реализация сборщика мусора – сложное занятие при создании функционального языка программирования, так как данных копится довольно много, и сложно понять, что же следует удалить.

2.8 Модульность

Модульная система языка программирования позволяет реализовывать программу, как совокупность небольших независимых блоков, не повторять написание часто встречающихся конструкций или функций, а разместить их в отдельном модуле. Реализация подобной особенности – сложная задача, так как требует написания специальных алгоритмов для разрешения зависимостей. Например, у нас есть базовый модуль, допустим “prelude”, но вдруг оказывается, что в текущей директории тоже есть подобный модуль, да и где-то в цепочке импортов. Что с этим делать? Как раз таки на этот вопрос отвечает модульная система языка, а не программист. Поэтому я попробовал написать самую базовую реализую подобной системы, которая бы удовлетворяла самым простым потребностям: не переписывать повторяющийся код и иметь возможность разделить исходный код на модули, находящиеся в текущей директории.

Ознакомиться с исходным кодом можно как всегда по ссылке:

https://github.com/geothecode/YAL/blob/master/yal/source/Module.hs.

2.9 Интерпретация

Интерпретация – это построчный анализ и обработка исходного кода программы. Интерпретатор – некоторый алгоритм, программа, выполняющая интерпретацию.

Для того чтобы работа с языком не была чисто теоретическим занятием, я реализовал простой интерпретатор для языка “YAL”, способный импортировать модули, подключать языковые расширения, вычислять выражения и сообщать о возникающих ошибках.

Исходный код находится по ссылке:

https://github.com/geothecode/YAL/blob/master/yal/source/Shell.hs.

Заключение

В заключение хотелось бы отметить, что наш язык[5] всё ещё требует серьёзных доработок:

- Kind система
- Ленивые вычисления с мемоизацией
- Сборщик мусора
- Доработка модульной системы
- и многое другое, что делает любой функциональный язык удобным и полезным к использованию

Всё, что мы реализовали – лишь малая часть того, что делает функциональный язык программирования хорошим языком. Но мы добились реализации всех изначально поставленных целей, даже сделали

[5] Исходный код реализованного языка находится по ссылке в репозитории: https://github.com/geothecode/YAL

слегка больше, что нельзя не отметить как успешное исследование. Подводя некий итог, я хотел бы отметить те понятия и навыки, которыми я овладел, выполняя исследование:

- Понял, насколько трудно реализовать полноценный функциональный язык программирования
- Научился основным приемам в реализации алгоритмов вывода типов
- Получил основное представление о многих терминах, используемых специалистами, работающими в этой сфере
- Освоился с базовыми принципами организации научного текста

Хотелось бы также отметить те области и теории, которые стали серьезной опорой при подготовке материала:

- Теория типов
- Теория языков программирования
- Ленивые вычисления
- Работа с памятью
- И, конечно же, функциональное программирование

Список литературы

Печатные издания:

- Теория типов
  - (IFL)[6] Implementing Functional Languages – Simon L. P. Jones, David R. Lester [1992, Prentice Hall, 296 стр.]
  - (TPL) Types and Programming Languages – Benjamin Pierce [2002, MIT Press, 680 стр.]
- Haskell
  - (LYH) Learn You A Haskell For A Great Good – Miran Lipovacha [Оригинал: No Starch Press, San Francisco; Русское издание: ДМК Пресс, Москва, 2017, 490 стр.]
  - (HID) Haskell In Depth – Виталий Брагилевский [Manning Publications Co., 2021, 664 стр.]

Интернет-ресурсы:

- Теория типов
  - Лекции по теории типов – Виталий Брагилевский: https://www.youtube.com/watch?v=_HYI7zjkrEs&list=PLvPsfYrGz3wuVAGhNf6-i7uafXg56oqM5&index=1
  - Write You A Haskell – Stephen Diehl [2015, CC BY-NC-SA 4.0]
- Haskell
  - What I Wish I Knew When I Was Learning Haskell – Stephen Diehl [2009]

- Теория языков программирования

[6] В скобках отмечено краткое название книги для создания удобных ссылок в тексте.

    - https://habr.com/ru/post/348874/ [Статья и перевод: Анатолий Ализар; Оригинал: Adrian Colyer]
- Общее
    - https://ru.wikipedia.org/wiki/%D0%94%D0%B5%D1%80%D0%B5%D0%B2%D0%BE_(%D1%81%D1%82%D1%80%D1%83%D0%BA%D1%82%D1%83%D1%80%D0%B0_%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D1%85
    - https://enterprisecraftsmanship.com/posts/what-is-functional-programming/ [Vladimir Khorikov]
    - https://ru.wikipedia.org/wiki/%D0%A2%D0%B5%D0%BE%D1%80%D0%B8%D1%8F_%D0%BC%D0%BD%D0%BE%D0%B6%D0%B5%D1%81%D1%82%D0%B2
    - https://ru.wikipedia.org/wiki/%D0%9C%D0%B5%D0%BC%D0%BE%D0%B8%D0%B7%D0%B0%D1%86%D0%B8%D1%8F
    - https://ru.wikipedia.org/wiki/%D0%9A%D0%B0%D1%80%D1%80%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5
    - https://www.wikiwand.com/ru/%D0%9B%D1%8F%D0%BC%D0%B1%D0%B4%D0%B0-%D0%B8%D1%81%D1%87%D0%B8%D1%81%D0%BB%D0%B5%D0%BD%D0%B8%D0%B5
    - https://www.wikiwand.com/ru/%D0%92%D1%8B%D0%B2%D0%BE%D0%B4_%D1%82%D0%B8%D0%BF%D0%BE%D0%B2
    - https://markkarpov.com/tutorial/megaparsec.html

1. Реализованный язык программирования не совсем является чистым, так как мы не имеем kind system, которая позволяла бы нам создать адекватное разделение чистых вычислений (pure) от нечистых (impure). К примеру, в правильной системе IO ::[7] * → *, у нас же – IO :: * .

[7] Имеет kind (::).