2.20. Функциональные языки программирования

1 Функциональные языки программирования

В языках функционального программирования основным конструктивным элементом является математическое понятие функции. Существует различия в понимании функции в математике и функции в программировании, вследствии чего нельзя отнести Си подобные языки к функциональным, использующим менее строгое понятие. Функция в математике не может изменить вызывающее её окружение и запомнить результаты своей работы, а только предоставляет результат вычисления функции. Программирование с использованием математического понятия функции вызывает некоторые трудности, поэтому функциональные языки, в той или иной степени предоставляют и императивные возможности, что ухудшает дизайн программы (например возможность безболезненных дальнейших изменений). Дополнительное отличие от императивных языков программирования заключается в декларативности описаний функций.

В качестве основных свойств функциональных языков программирования обычно рассматриваются следующие:

Функции как значения Функции — объекты первого класса, то есть с ними можно проводить различные операции и вычисления. функции могут быть переданы другим функциям в качестве аргумента или возвращены в качестве результата.

Функции высших порядкок Функции, принимающие функциональные аргументы, называются функциями высших порядков или функционалами.

Рекурсия Большинство функциональных языков реализуют оптимизацию хвостовой рекурсии, что позволяет записывать многие алгоритмы в естественной рекурсивной форме, не упираясь в переполнение стека.

Сборка мусора Позволяет отказаться от низкоуровневой работы с памятью, тем самым уменьшая количество ошибок.

Отсутствие побочных эффектов В чистом функциональном программировании оператор присваивания отсутствует, объекты нельзя изменять и уничтожать, можно только создавать новые путем декомпозиции и синтеза существующих. О ненужных объектах позаботится встроенный в язык сборщик мусора. Благодаря этому в чистых функциональных языках все функции свободны от побочных эффектов.

Строгая типизация Строгая типизация обеспечивает безопасность — в функциональных языках большая часть ошибок может быть исправлена на стадии компиляции, поэтому стадия отладки и общее время разработки программ сокращаются. Вдобавок к этому строгая типизация позволяет компилятору генерировать более эффективный код и тем самым ускорять выполнение программ. В большинстве строго типизированных функциональных языках встроен специальный механизм, позволяющий компилятору определять типы констант, выражений и функций из контекста. Этот механизм называется механизмом вывода типов. Известно несколько таких механизмов, однако большинство из них являются разновидностями модели типизации Хиндли-Милнера, разработанной в начале 80-х годов XX века. Таким образом, в большинстве случаев можно не указывать типы функций.

Модульность Механизм модульности позволяет разделять программы на несколько сравнительно независимых частей (модулей) с чётко определёнными связями между ними. Тем самым облегчается процесс проектирования и последующей поддержки больших программных систем.

Эффективные способы работы со списками При наличии сборки мусора списки — гибкая и удобная структура данных, допускающая эффективную обработку в функциональном стиле.

Отложенные (ленивые) вычисления В традиционных языках программирования (например, C++) вызов функции приводит к вычислению всех аргументов. Этот метод вызова функции называется вызов-по-значению. Если какой-либо аргумент не использовался в функции, то результат вычислений пропадает, следовательно, вычисления были произведены впустую. В каком-то смысле противоположностью вызова-по-значению является вызов-по-необходимости. В этом случае аргумент вычисляется, только если он нужен для вычисления результата. Примером такого поведения можно взять оператор конъюнкции всё из того же C++ (&&), который не вычисляет значение второго аргумента, если первый аргумент имеет ложное значение. Языки, использующие отложенные вычисления, называются нестрогими.

Перечислим основные языки функционального программирование.

Lisp Берет истоки в λ-исчислении Алонзо Чёрча. Динамически типизируемый. Содержит массу императивных свойств, однако в общем поощряет именно функциональный стиль программирования. Использует единообразный и простой синтаксис, основанный на списках, вследствие чего может оперировать с кодом как с данными и наоборот. Это позволяет реализовать мощнейшее средство Лиспа, не имеющее аналогов в других современных языках программирования, — макросы — функции над кодом программы, исполняемые компилятором при его обработке, что позволяет легко создавать специальные встроенные языки программирования для конкретных задач (DSEL, domain-specific embedded languages). Основные диалекты:

- **Common Lisp** В нем реализована первая и мощнейшая на сегодняшний день объектная система — CLOS (Common Lisp Object System). Основная проблема — устаревший стандарт, который вряд ли будет обновляться, и множество реализаций, каждая из которых имеет свои особенности. Основные реализации — Clisp, SBCL, Allegro CL, LispWorks.
- **Scheme** диалект Лиспа, созданный специально для использования при обучении и научной работы в области computer science. Имеет компактный и простой стандарт. Основные реализации - DrScheme/MzScheme, Chicken, Guile.
- **Elisp** диалект Лиспа, используемый в редакторе Emacs. Обладает существенным недостатком — имеет только переменные с динамической областью видимости, переменные с лексической областью видимости отсутствуют. Это затрудняет эффективную компиляцию языка и усложняет программы.
- **Clojure** Современный перспективный диалект Лиспа, использующий JVM. Имеет мощную объектную систему, поддерживающую мультиметоды, мультипотоковое программирование с разделяемой памятью используя STM (software transactional memory), отличную интеграцию с Java. Основной недостаток — подвержен ограничениям, накладываемым JVM.

ML Сильно типизированный язык со статическим контролем типов, аппликативным выполнением программ, развитой полиморфной системой типов, параметризуемыми модулями и автоматическим выводом типов. Содержит некоторые императивные свойства, такие как ссылки на изменяемые значения и поэтому не является чистым. Основные диалекты:

- **Standard ML** Современная реализация ML. Среди других языков программирования уникален тем, что имеет формальную спецификацию. Основные реализации: MLton, Standard ML of New Jersey, Moscow ML.
- **OCaml** Самый распространенный диалект ML. Включает средства для поддержки объектно-ориентированной парадигмы программирования. Имеет эффективный оптимизирующий компилятор в машинный код.
- **F#** Структура F# во многом схожа со структурой OCaml с той лишь разницей, что F# реализована поверх библиотек и среды исполнения .NET.

Haskell Является одним из самых распространённых нестрогих языков программирования. Имеет очень развитую систему типизации, автоматический вывод типов, является чистым языком, использует параметрический полиморфизм, параметризм классов типов и концепцию монад для эмуляции императивных вычислений. Основная реализация — GHC (Glasgow Haskell Compiler).

Erlang Заточен под написание программ для различного рода распределенных систем. Язык использует модель легковесных процессов и включает в себя средства порождения параллельных процессов и их коммуникации с помощью посылки асинхронных сообщений. Чистый язык с динамической типизацией.

Scala, Nemerle Гибридные языки со статической типизацией, сочетающие возможности функционального и объектно-ориентированного программирования. Scala реализована поверх JVM и обладает хорошей поддержкой компонентного ПО. Nemerle использует .NET и имеет мощную систему метапрограммирования.

2 Функции высших порядков

Функции высших порядков — это функции, аргументами или результатами которых также являются функции. В традиционных языках программирования такие функции часто бывает невозможно описать.

В функциональном программировании, однако, функции являются значениями "первого класса так что подобные описания не только возможны, но и достаточно часто встречаются. В частности, функция, определяющая суперпозицию двух других функций, не только может быть легко описана средствами языка, но и является одним из самым важных инструментов построения программ, и поэтому содержится в языке в виде стандартной операции.

Самой распространенным примером функции высшего порядка является, видимо, функция map, применяющая функцию-аргумент ко всем элементам данного списка. Рассмотрим ее реализацию на языке Haskell.

```
map :: (a -> b) -> [a] -> [b]
map _ []     = []
map f (x:xs) = f x : map f xs
```

Если некоторая функция предназначена только для того, чтобы передать ее в качестве аргумента другой функции, то не имеет смысла давать ей постоянное имя и определять ее тип явно. В этом случае можно использовать так называемое лямбда-выражение, которое задает образцы для аргументов функции и правые части ее уравнений, но не связывает с этой функцией никакого имени, а тип такой функции определяется из контекста. Таким образом лямбда-выражение представляет собой функциональное значение, изображает безымянную функцию. Например, можно вычислить при помощи анонимной функции квадрата каждого элемента списка: `map (\x -> x * x) [1, 2, 3, 4]`.

Функции высших порядков можно определять, чтобы строить новые функции на основе других. Наверное, одной из самых распространенных операций над функциями является их суперпозиция (или композиция). Такая операция позволяет по двум функциям f и g получить новую функцию $f\dot{g}$ такую, что результат ее применения к аргументу $f\dot{g}x$ будет тем же самым, что и результат последовательного применения функций f и g: $f(gx)$. В отличие от традиционных языков программирования, в которых описать подобную функцию невозможно, в языке Haskell такое описание не составляет никакого труда:

```
comp :: (b -> a) -> (c -> b) -> (c -> a)
comp f g = \x -> f (g x)
```

Продемонстрируем возможность возвращать из функции другую функцию и поддержку функциональными языками замыканий на примере создания объекта-счетчика, реализованного в виде функции на Common Lisp.

```
(defun make-counter ()
  (let ((counter 0))
    (lambda ()
      (incf counter)
      counter)))

(setf c (make-counter))
(format t "~a ~a ~a"
        (funcall c)
        (funcall c)
        (funcall c))
```

После запуска программы мы получим на экране `1 2 3`.

3 Нормальный и аппликативный порядок вычислений, ленивые вычисления

Понятия нормального и аппликативного порядка вычислений пришли из теории λ-исчисления. Рассмотрим выражение, задающее применение функции к аргументу $(\lambda x.e1)e2$, где $e1$, $e2$ — также выражения. Можно придерживаться двух возможных порядков вычисления этого выражениюю. Способ, который используется в большинстве языков программирования, — энергичное вычисление, который соответствует методу «вычисление аргументов, затем применение процедуры». В этом случае сначала будет вычислено выражение $e2$, а затем с использованием полученного аргумента будет проведена β-редукция. Такой способ называется *аппликативным порядком вычислений* (applicative-order evaluation). Альтернативный метод «полная подстановка, затем редукция» известен под названием *нормальный порядок вычислений*(normal-order evaluation). В этом случае сразу будет выполнена подстановка $e2$ в $e1$.

Нормальный порядок вычислений является основой для ленивых вычислений в современных функциональных языках. Ленивые вычисления или отложенные вычисления (lazy evaluation) —– концепция, согласно которой вычисления следует откладывать до тех пор, пока не понадобится их результат. В этом способе, как и при передаче аргумента по наименованию, фактически в процедуру передается функция для вычисления значения или адреса фактического аргумента. Однако, при первом же обращении к аргументу изнутри процедуры вычисленное однажды значение (или адрес) запоминаются. При последующих обращениях к этому же аргументу используется уже вычисленное однажды значение. В языке Haskell по умолчанию везде используются ленивые вычисления, в языке OCaml, наоборот, по умолчанию используются энергичные вычисления, а для применения ленивых вычислений необходимо использовать ключевое слово `lazy`.

Ленивые вычисления позволяют сократить объем вычислений, результаты которых не будут использованы. Тем самым позволяют программисту описывать только зависимости функций друг от друга и не следить за тем, чтобы не осуществлялось «лишних вычислений». Кроме того, ленивые вычисления серьезно расширяют выразительные способности языка. Так, использование ленивых вычислений позволяет записывать в удобном виде и применять функции, обрабатывающие бесконечные потоки данных (streams), а также позволяют работать с такими структурами данных, как бесконечные списки. Так, на языке Haskell можно получить бесконечный ряд чисел Фибоначчи, используя ленивые вычисления и функции высших порядков, следующим образом: `fib = 0:1:zipWith (+) fib (tail fib)`.

4 Эмуляция императивного поведения. Язык Haskell

Эмуляция императивного поведения в чистом ленивом функциональном языке сопряжена со следующими проблемами:

- Так как язык чистый, то функции не могут иметь побочных эффектов. Но любой ввод-вывод, по сути, является побочным эффектом, а язык без механизма ввода-вывода будет бесполезной «вещью в себе».
- Ленивость языка означает, что невозможно гарантировать порядок вычислений. В то же время нам необходимо это делать, так как при том же вводе-выводе порядок вычисления важен.

В языке Haskell для эмуляции императивного поведения используются монады. Интуитивно монады можно представлять себе как контейнерный тип. Математически монада задается двумя операциями:

```
class  Monad m  where
    (>>=)  :: m a -> (a -> m b) -> m b
    return :: a -> m a
```

Операция `return` позволяет «положить» значение в «контейнер», который представляет собой монада. Операция `(>>=)` — операция связывания. комбинирует монадическое значение m a, содержащее объект типа a, с функцией, которая оперирует значением типа a и возвращает результат типа m b.

Операции `return` и `(>>=)` должны удовлетворять следующим требованиям:

1. `return a >>= k  ==  k a`
2. `m >>= return  ==  m`
3. `m >>= (\x -> k x >>= h)  ==  (m >>= k) >>= h`

Точное значение операция связывания конечно же зависит от конкретной реализации монады. Так, например, монада ввода-вывода IO (которую можно представить себе как контейнер, содержащий «внешний мир») определяет операцию `(>>=)` как последовательное выполнение двух её операндов, а результат выполнения первого операнда последовательно передается во второй. Например, для чтения Для двух других встроенных монадических типов (списки и Maybe) эта операция определена как передача нуля или более параметров из одного вычислительного процесса в следующий.

Так как списки являются монадами, они являются отличным материалом, на котором можно рассмотреть практическое применение механизма монад. Для списков операция связывания обретает смысл в соединении вместе набора операций, производимых над каждым элементом списка. При использовании со списками сигнатура операции `(>>=)` приобретает следующий вид: `(>>=) :: [a] -> (a -> [b]) -> [b]` Это обозначает, что дан список значений типа a и функция, которая проецирует значение типа a на список значений типа b. Связывание применяет функцию к каждому элементу данного списка значений типа a и возвращает полученный список значений типа b. Приведем полное определение монадических операций для списков:

```
return x = [x]
xs >>= f = concat (map f xs)
```

Все операции ввода-вывода используют монаду IO. Рассмотрим, как можно записать простейшую программу используя введенные операции связывания и стандартные функции `putStrLn :: String -> IO ()` и `getLine :: IO String`:

```
main = putStrLn "Please enter your name: " >>= \_ ->
       getLine >>= \name ->
       putStrLn ("Hello, " ++ name ++ ", how are you?")
```

Применение монад в функциональных языках — это по существу возвращение к императивности. Ведь операция связывания (`>>=`) предполагает последовательное выполнение связанных выражений с передачей или без результатов вычисления. Т.е. монады — это императивное ядро внутри функциональных языков. С одной стороны это идёт в разрез с теорией функционального програмирования, где отрицается понятие императивности, но с другой стороны некоторые задачи решаются только при помощи императивных принципов. И опять же, Haskell предоставляет удивительную возможность по генерации списков, но это только благодаря тому, что сам тип "список"выполнен в виде монады.