План лекции

- Лирическое отступление: финансы и ФП
- Свёртки на синтаксических деревьях
- Обобщение свёрток: катаморфизмы
- Почему ФП?

Barclays Capital

- Barclays Capital – инвестиционный банк
- Предоставляет брокерские услуги, анализ рынков и т.д.
- Наш проект: создание формального языка для описания деривативов

Пример деривативa

- У Петра имеется 100 акций компании Apple
- Цена 1 акции сегодня: \$5
- Он может продать их сейчас или продать позже
- Пётр любит Apple и надеется на то, что акции подорожают, но не слишком уверен в этом
- Как ему застраховаться от возможных убытков?
- Купить у нас деривативв!

Пример деривативa

- Банк может заключить с Петром контракт, что
  1. если акции подешевеют – банк выплатит разницу в цене
  2. в противном случае – банк ничего не выплачивает
- Математически:
  `max(S0 - S1, 0)`
- Пётр может воспользоваться этим контрактом через год.
- Таким образом, если через год цена 1 акции составит 2 доллара, Пётр будет застрахован от убытков и сможет продать акции по той же цене, что и в день заключения контракта.
- Однако за эту услугу “страховки” Петру придётся заплатить какую-то сумму, чтобы подписать такой контракт.
- Деривативв – это и есть контракт такого рода, зависящий от цены базового актива (цены акций, валют и т.д.).

Опционы

- Базовые активы – это что-то более конкретное (цена валюты раньше соответствовала “золотому запасу”, акция отражает заинтересованность во владении часть компании)
- Деривативы – более абстрактное, это просто функция от базовых активов. Цена деривативов является производной от цен активов (отсюда название).
- Дериватив, предложенный Петру, называется “опцион”
- Опционы могут быть сложнее (“экзотические опционы”), например:
- Вычислить среднее значение цены за последний год и выплатить его
- Сравнить цены нескольких различных акций и выплатить минимум
- Выплатить взвешенное среднее цен нескольких акций

Пример в коде

- Наш язык описывает такие контракты и выплаты по ним в виде, очень похожем на язык программирования.
- Функция для описания выплаты Петру

```
payout :: Market -> Double
payout market =
  let s0 = observe market "28-10-2011" "AAPL"
      s1 = observe market "28-10-2012" "AAPL"
  in max (s0 - s1) 0
```

Описание контрактов в коде

- В чём преимущество описания в виде программы?
  1. Гораздо лучше бумажного документа со словесным описанием, который может допускать неоднозначности
  2. Поскольку это программа, её можно выполнить и точно посчитать выплату
- Недостатки
  - Выполнить можно только когда наступит дата выплаты (цены акций - случайные величины)

Определение стоимости опциона

- Однако, для того, чтобы продавать опционы нужно оценивать выплату заранее.
- Для того, чтобы оценить выплату ещё до его продажи нужно примерно знать какими будут значения случайных величин.
- Одно из решений: симуляция случайных величин методом Монте-Карло.
  - каждая случайная величина имеет свою модель
  - этим занимается отдел аналитиков
- Мы можем запустить 100000 раз симуляцию со случайной ценой акции, распределённой согласно своей модели, для каждого раза посчитать выплату, усреднить и попросить у Петра немного больше – за сервис
- Теоретически, никто не мешает Петру проделать это самому, чтобы оценить риск, но как правило у него нет квалификации или нужных инструментов

Инструменты

- У нас такие инструменты есть!
- FPF. Functional Payout Framework. Написан на Haskell.
- Включает в себя язык описания деривативов.
- Выполняет такие функции:
- Генерация кода для оценки выплат (pricing instructions) по контрактам
- Генерация описаний контрактов в виде формул
- Различные анализы контрактов
- Многое другое

Пример описания контракта

- Код скриптов представляет собой код на Haskell
- Финансисты не очень хорошо читают Haskell, им понятнее форумы, потому мы генерируем описания контрактов в TeX
- См. отдельный слайд

Извлечение информации

- Важная информация, которую мы можем получить из контракта-скрипта:
- Даты наблюдения
- Множество базовых активов деривативa
  - например названия задействованных в контракте акций
- Точки разрыва функции выплаты
  - чтобы вовремя на них реагировать

Как мы можем получить эту информацию?

- Синтаксический разбор скрипта
- Построение синтаксического дерева
- Анализ дерева с помощью свёрток

Синтаксические деревья

- **Синтаксическое дерево** – это промежуточное представление кода программы в компиляторах языков программирования.
- В процессе проверки синтаксического разбора программы парсер строит синтаксическое дерево, которое потом подвергается различным манипуляциям для оптимизации, проверки типов и т.д.
- Вспомните лабораторную по синтаксическим деревьям регулярных выражений.

Свёртки для алгебраических типов данных

- Суть *свёртки* определяется названием: она **сворачивает** сложную структуру данных в некоторое значение. Так можно решить много разных задач, в том числе и поставленные выше.
- Изобразим на доске пример сложной структуры в виде дерева, где каждый узел – это конструктор.
- Рядом изобразим дерево аналогичной структуры, только с операциями в узлах. Каждому конструктору соответствует отдельная операция.

Синтаксические деревья

- Мы будем рассматривать упрощённую версию дерева из языка, определённого в FPF.
- Тип такого дерева:

```
type expr =
    | Const of int
    | Var of string
    | Add of expr * expr
    | Mult of expr * expr
    | Observe of date * string
```

Примеры выражений

- $observe"28-10-2011AAPL"$

```
Observe(Date "28-10-2011", "AAPL")
```

- $2+3$

```
Add(Const 2, Const 3)
```

- $s0+s1$

```
Add(Var "s0", Var "s1")
```

- $0+(1*x))$

```
Add(Const 0, Mult(Const 1, Var "x"))
```

Задача

- Написать функцию для расчёта выплаты, используя дерево.
  - на входе: синтаксическое дерево контракта (выражение)
  - на выходе: выплата (число)
- Это – задача свёртки, по сути нам нужно свернуть дерево по неким правилам в одно значение.
- Теперь нужно определить операции, соответствующие разным конструкторам типа, которые будут решать задачу.

Свёртки и конструкторы

- Вспомним о свёртках на списках, чтобы уяснить важный момент
- Какие аргументы у функции свёртки для списков?
  1. Значение на которое мы заменяем пустой список.
  2. Функция для комбинирования элемента списка с результатом свёртки хвоста списка.
- Сравните вычисление правой свёртки на списке и структуру самого списка:

$$x_1 \oplus (x_2 \oplus (x_3 \oplus (\ldots \oplus (x_n \oplus base))))$$

$$x_1 :: (x_2 :: (x_3 :: (\ldots :: (x_n :: []))))$$

- Видно, что правая свёртка это по сути замена конструктора списка (`::`) на некую операцию. При этом финальный пустой список заменяется на некое значение `base`.

Свёртки и конструкторы

- Список – это алгебраический тип данных с двумя конструкторами

```
type 'a list =
  | Nil
  | Cons of 'a * ('a list)
```

- Первый аргумент `fold` соответствует пустому списку (конструктор `Nil`)
- Второй аргумент соответствует голове и свёртке хвоста (конструктор `Cons`)
- Важно: свёртка – это замена конструкторов типа на заданные функции
- Таким образом теперь нам надо заменять конструкторы дерева на некие функции

Преобразование типа данных в “контекст”

- Можно решить данную задачу с помощью рекурсивно определённой функции. Но можно сделать это более элегантно, задав простые и нерекурсивные способы комбинирования значений, а всю рекурсию спрятать в оператор свёртки.
- Введём понятия “контекста свёртки”. В контексте будут храниться уже вычисленные значения предыдущих свёрток.
- Чтобы получить тип контекста применим преобразование к типу `expr`, чтобы получить новый тип данных.

Преобразование

Было:

```
type expr =
    | Const of int
    | Var of string
    | Add of expr * expr
    | Mult of expr * expr
    | Observe of date * string
```

1. Вводим новый параметр типа a
2. Заменяем рекурсивные вхождения expr в оригинальном типе на параметр a
3. Переименовываем конструкторы, добавляя `Context`.

Стало:

```
type 'a exprContext =
    | ConstContext of int
    | VarContext of string
    | AddContext of 'a * 'a
    | MultContext of 'a * 'a
    | ObserveContext of date * string
```

Функции на контексте

- Наш контекст – типизированный, в нём могут храниться определённые значения типа a, а именно – промежуточные результаты.
- Теперь определим функцию на контексте, определяющую свёртку
- Она будет иметь тип `'a exprContext -> a`. Эта функции будет нерекурсивной и её достаточно для того, чтобы полностью определить свёртку.
- Для нашей задачи результатом свёртки будет `int`, т.к. результат вычисления выражения – число, а именно результат определяет тип a в контексте.

Функции на контексте

- Таким образом, нам надо задать функцию вида `'int exprContext -> int`

```
// eval : 'int exprContext -> int
let eval expr =
    match expr with
    | ConstContext x          -> x
    | VarContext s            -> 42 // lookup in env
    | AddContext (x, y)       -> x + y
    | MultContext (x, y)      -> x * y
    | ObserveContext (d, a) -> lookup d a market
```

- Здесь мы не делаем рекурсивных вызовов, а просто описываем как комбинировать элементы, считая что предыдущие вызовы у нас уже вычислены. Можно сказать, что эта функция знает как свернуть один уровень структуры данных.
- Изобразим на доске картинку, которая сравнит функцию на деревьях и функцию на контекстах.
- Функция `eval` нерекурсивна!

Катаморфизм

- Как теперь “запустить” эту функцию, чтобы она приняла дерево и выдала результат?
- Для этого нам нужно преобразовать функцию `f`, которая определяет свёртку, в реальную свёртку. То есть у нас есть определение того, как свернуть один уровень дерева, и нам нужно свернуть всё дерево, обладая только этой функцией и знанием о типе дерева. Это делается с помощью функции такого рода:

```
// toFold : ('a exprContext -> 'a) -> expr -> 'a
let rec toFold ctx f =
    match ctx with
    | Const x          -> f (ConstContext x)
    | Var   s          -> f (VarContext s)
    | Add  (x, y)      -> f (AddContext (toFold f x, toFold f y))
    | Mult (x, y)      -> f (MultContext (toFold f x, toFold f y))
    | Observe (d, a) -> f (ObserveContext(d, a))
```

Катаморфизм

- Именно в этой функции “зашита” рекурсия, однако прелесть в том, что эта функция может быть обобщена на любой рекурсивный тип данных (практически любой) и её не нужно будет создавать заново.
- Такое обобщение свёрток на алгебраические структуры данных называется “катаморфизмом”.
- “ката” – по-гречески означает “вниз”. Однокоренное слово “катастрофа”, то есть “разрушение”. Катаморфизм как бы дробит, разрушает структуру данных, преобразовывая её в какое-то единое значение.
- Можно переименовать функцию `toFold` в `cata`

Решение задачи про вычислитель

- Соответственно, финальная функция, которая будет решать нашу задачу выглядит так:

```
// e = 2 + 3 * 4
let e = Add(Const 2, Mult(Const 3, Const 4))
let solve e = cata eval e
let main = printfn "%A" (solve e)
// output: 14
```

- Важно, что `f` – нерекурсивна, вся рекурсия спрятана в `toFold`.

Задача оптимизации выражений

- Предположим нам нужно написать оптимизатор выражений.
- К примеру у нас есть выражение `1 * (x + 0)`. Вычислить мы его не можем, поскольку неизвестен `x` однако мы можем оптимизировать, выбросив умножение на 1 и сложение с нулём, т.к. они не меняют результат.

Решение

- Для её решения определяем ещё одну функцию для контекста.
- Теперь параметром a в контексте будет не число, а `expr`, поскольку результатом нашей функции является выражение.
- Функция будет задавать наши правила оптимизации.

```
// optimizer : expr exprContext -> expr
let optimizer e =
  match e with
    | ConstContext x              -> Const x
    | VarContext s                -> Var s
    | AddContext (Const 0, y)     -> y
    | AddContext (y, Const 0)     -> y
    | AddContext (x, y)           -> Add (x, y)
    | MultContext (Const 1, y)    -> y
    | MultContext (y, Const 1)    -> y
    | MultContext (x, y)          -> Mult (x, y)
    | ObserveContext (d, a)       -> Observe (d, a)
// expr for '1 * (x + 0)'
let e2 = Mult(Const 1, Add(Var "x", Const 0))
let main = printfn "%A" (toFold optimizer e2)
// output: Var "x"
```

Катаморфизм на списках

- Можно провести аналогичную операцию со введением контекста для списков, чтобы показать как будет выглядеть свёртка в варианте с контекстом.

```
// type for list fold context
type 'a listContext =
   | NilContext
   | ConsContext of int * 'a

// function which matters (defines combination,
// one case for one constructor)
sumC NilContext          = 0
sumC (ConsContext x s) = x + s

// sum as catamorphism
sum = cata sumC
```

Запись свёртки на деревьях через foldExpr

- Ещё один вариант определения свёрток на синтаксических деревьях: написание функции, аналогичной функции fold для списков.

```
// foldExpr : (a->a->a) -> (a->a->a) -> (int->a) ->
// -> (string->a) -> (date->string->a) -> expr -> a
let foldExpr add mult constf var obs e =
    let rec go e =
      match e with
        | Add (e1, e2)   -> add (go e1) (go e2)
        | Mult (e1, e2)  -> mult (go e1) (go e2)
        | Const c        -> constf c
        | Var name       -> var name
        | Observe (d, a) -> obs d a
    go e
```

- Здесь, как и в случае с `fold` на списках, мы задаём по одной функции для каждого из конструкторов.
- Тип функции foldExpr довольно ужасен (много сложных параметров)

Ещё один пример вычислителя

Тогда вычислитель будет выглядеть так:

```
// yet another evaluator
let eval2 env market e =
    let add x y  = x + y
    let mult x y = x * y
    let constF c = c
    let var s    = lookup s env
    let obs d a  = lookupMarket d a market
    foldExpr add mult constF var obs e

// e = 2 + 3 * 4
let e = Add(Const 2, Mult(Const 3, Const 4))
let main = printfn "%A" (eval2 emptyEnv getMarketForToday e)
// output: 14
```

Сравнение `foldl` на списках и `cata`

- Зачем это нужно? В чём преимущество использования контекста перед обычным использованием `foldr/foldl` для списков?
  1. В функции `foldr/l` каждому типу конструктора соответствовал свой аргумент (базовый элемент для пустого списка, комбинирующая функция – для непустого), таким образом количество аргументов соответствует количеству конструкторов. Во многих случаях это неудобно, т.к. конструкторов может быть очень много (например в реальном синтаксическом дереве). Использовать функции с десятками аргументов непрактично: это нечитабельно, сложно поддерживать, нужно менять функцию каждый раз при добавлении нового конструктора.
  2. При создании нового типа данных для него пришлось бы создавать свою функцию `foldX`, которая задавала бы определённую схему рекурсивной свёртки. В нашем же случае всё, что нам нужно – это задать тип данных для контекста, всё остальное может быть выведено автоматически с помощью "комбинатора неподвижной точки" над типами данных. Как именно – будет рассмотрено в следующем семестре. Таким

Barclays Capital и ФП

Почему функциональное программирование и Haskell?

- Хорошо соответствует предметной области: выплаты по деривативам представляются в виде математических функций (выходы чётко соответствуют входам, нет сайд-эффектов): их можно сочетать, передавать в функции высшего порядка.
- Легко сделать поддержку функций высшего порядка в скриптах. Это удобнее для скриптеров, позволяет более кратко описывать важные концепции.
- Сопоставление с образцом (pattern matching) удобно для преобразования деревьев. Операции на деревьях – одни из основных в компиляторах и всяких преобразователях или генераторах кода.
- Статическая типизация и отсутствие сайд-эффектов позволяет писать “код, в который можно верить”.
- Более гибкий код, иногда даже слишком гибкий! Слабое связывание компонентов – легко делать рефакторинги.

Приходите к нам!

- Возможен internship (летняя практика) для талантливых студентов.
- Есть вакансии для Haskell-программистов и скриптеров.
- Знание Haskell желательно, но не обязательно, опыт с F# подходит.

Задачи

Несколько простых задач.

1. Реализовать катаморфизм на синтаксическом дереве, который возвращает множество переменных, использованных в выражении.
2. Разработать алгебраический тип и свёртки для следующих задач.
   2.1. У нас есть дерево документа, к примеру этой презентации. Элементами дерева могут быть:
   - параграф
   - картинка
   - ссылка
   - текст

   В каждом параграфе может быть произвольное количество картинок, подпараграфов, текста, ссылок и т.д.
   Задача состоит в том, чтобы извлечь все ссылки для того, чтобы вывести их в конце презентации. Напишите для этого катаморфизм.
   2.2. Напишите катаморфизм, который удалит все картинки из документа.

Контакты

- Иван Веселов
- Ivan.Veselov@BarclaysCapital.com
- veselov@gmail.com

Полезные ссылки

1. “Commercial Uses: Going functional on exotic trade” – статья про FPF
   http://arbitrary.name/papers/fpf.pdf
2. Набор статей о катаморфизмах в F# от одного из ведущих программистов команды F#
   http://lorgonblog.wordpress.com/2008/04/05/catamorphisms-part-one/
3. “Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire”
   Одна из первых статей про морфизмы в контексте программирования (сложная)
   http://citeseer.ist.psu.edu/viewdoc/summary?doi=10.1.1.41.125

Спасибо за внимание!

Задавайте вопросы.