ФП 2.0 [1][2][3]

осень 2025

Содержание

[1] Автор Андрей Стоян (`andrey.stoyan.csam@gmail.com`).

[2] Спасибо Илье Колегову за первое внимательное прочтение и кучу комментариев. Спасибо моим студентам, которые своим интересом к предмету сделали эту работу возможной.

[3] Версия от 14 января 2026 г., 11:37.

Введение

Многие сложные концепции в дизайне языков и программ могут быть поняты как частные случаи некоторых простых фундаментальных принципов, которые, как правило, считаются общеизвестным фольклором, не требующим дополнительных пояснений. Однако, сложность в том, что эти знания рассеяны по книгам, статьям и "культовым" блог-постам, и требуется довольно много времени и сил для восстановления целостной картины.

Цель данного курса — собрать в одном месте такие фольклорные знания и организовать их в некоторую систему. Курс будет явным образом опираться на классические работы, исследующие принципы построения языков, и помогать в их изучении. Просмотр упоминаемых статей является важной частью самостоятельной работы в рамках курса.

Под функциональным программированием, вынесенным даже в заголовок курса, понимается трепетное отношение к понятию эффекта, которое в ФП, в отличие от других школ мысли, не считается аксиоматической данностью, но предметом для изучения, сознательного конструирования и аккуратного обращения. Этот подход оказывается очень полезным для изучения языков, построения могущественных языковых конструкций, а так же является основой для продуктивного стиля программирования. Кроме того, функциональные языки сравнительно просты, в результате чего новые идеи и подходы нередко зарождаются в них и распространяются далее.

В качестве основного языка курса выбран Haskell, так как он, с одной стороны, воплощает в себе многие концепции, часто доведенные до некоторого логического завершения, и достаточно могуществен для кодирования других. С другой стороны, всё ещё является прикладным промышленным языком программирования.

В связи с широтой контекста, данный курс не всегда является глубоким. Так, детали реализации в GHC или теор-категорные основания вещей могут даваться в общем виде и без конкретики. В то же время, в плоскости языкового дизайна через оптику функционального программирования курс пытается быть максимально подробным.

Таким образом, данный курс может быть полезен тем, кто интересуется дизайном языков и красивыми обобщениями программистских концепций, хочет улучшить свои навыки проектирования API, или планирует вести практическую деятельность на функциональных языках.

Пререквизитом к прохождению курса является знание основ функционального программирования: алгебраических типов данных, паттерн-матчинга, свёрток, параметрического полиморфизма, классов типов, базовых монад. Дополнительно будет полезным умение читать типовые дроби, знакомство с полиморфным λ-исчислением и кодированием Чёрча.

1 Воспоминания о ФП

В этом разделе мы вспомним основные концепции функционального программирования и языка Haskell.

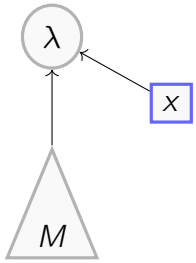
1.1 Термы и редукция

В ФП программы представляют собой выражения. Выполнение программ — редукция таких выражений до более "простых". Выражения можно представлять как в виде линейной записи символов, так и в виде дерева, для понимания которого не требуется знания вспомогательных правил ассоциативности и проч.

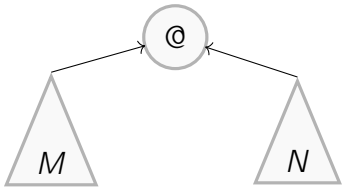
Простейший функциональный язык — λ-исчисление. Выражения в нём называются λ-термами, которые состоят из вершин трёх видов (V — множество валидных идентификаторов, Λ — множество λ-термов):

Переменные $x \in \Lambda$, если $x \in V$

Абстракция $(\lambda x.\, M) \in \Lambda$, если $x \in V, M \in \Lambda$



Аппликация $(M\ N) \in \Lambda$, если $M \in \Lambda, N \in \Lambda$



В произвольном выражении можно заменить некоторый его фрагмент на формальный параметр, который должен быть задекларирован выше по дереву с помощью специальной вершины λ. Вместо формального параметра можно в дальнейшем подставлять различные конкретные параметры с помощью вершины-аппликации @, то есть переиспользовать это выражение для различных целей (например, см. рис. 1). Редукция как раз определяется как следующее правило переписывания: ищется применение λ-функции к аргументу и в её тело осуществляется подстановка аргумента во все свободные вхождения переменной, связанной лямбдой (рис. 2).

1.2 Типы

Программное обеспечение — это сложно. Поэтому постоянно и неизбежно в программах возникают ошибки. Их можно искать, в том числе, статически, то есть без запуска программы. Одним из видов статического анализа является анализ типов.

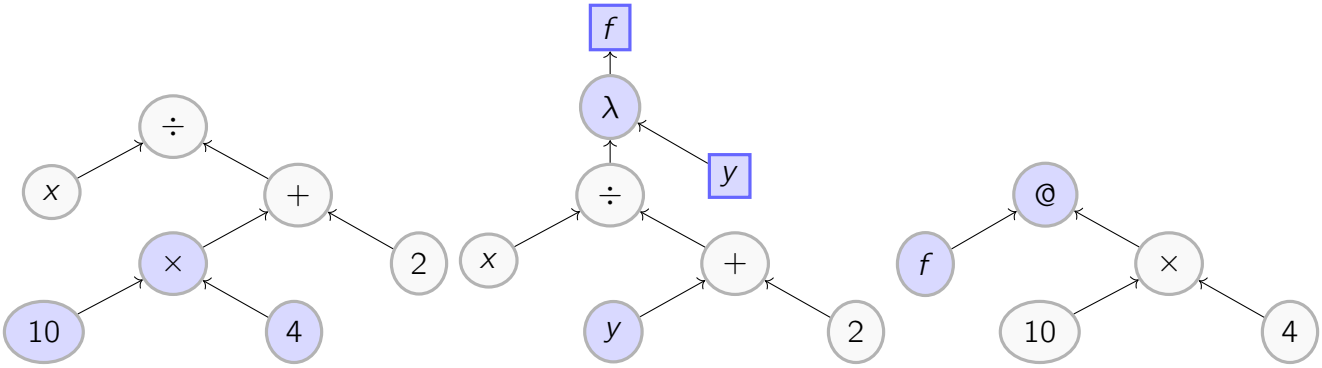


Рис. 1: Выражение с помощью λ вершины преобразуется в функцию одного аргумента.

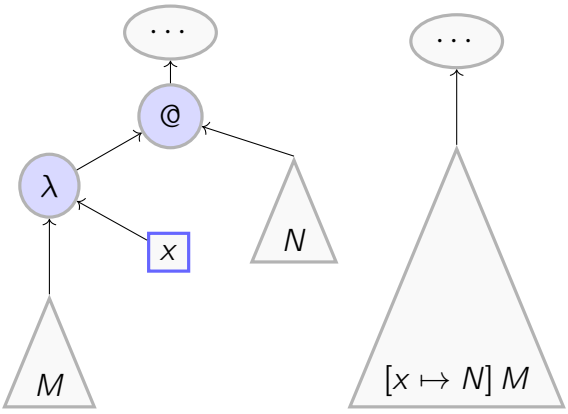


Рис. 2: Редукция переписывает дерево путём подстановки конкретного аргумента вместо формального параметра.

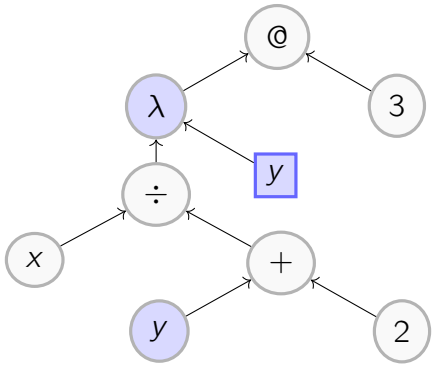


Рис. 3: Дерево соответствующее выражению $(\lambda y.\, x \div (y + 2))\ 3$.

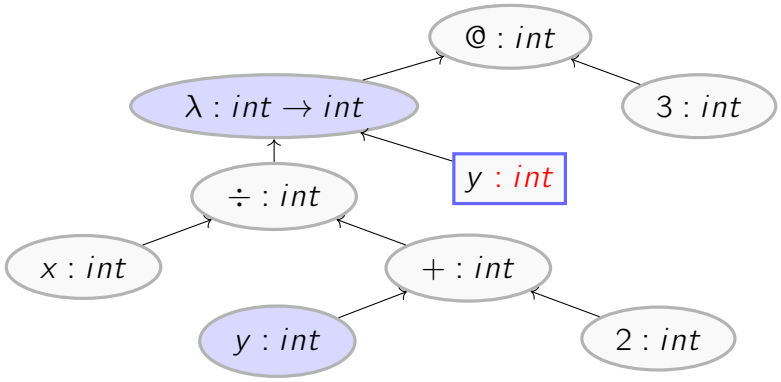


Рис. 4: Дерево выражения $(\lambda y.\, x \div (y + 2))\ 3$ после приписывания типовых меток.

Идея анализа типов состоит в том, что каждой вершине дерева программы мы пытаемся приписать некоторую синтаксическую метку по определённым правилам. Если таким образом каждой вершине можно приписать метку, то мы считаем, что программа проходит проверку типов, и она “хорошая”. Например, на рисунке 3 представлено выражение, а на рисунке 4 каждой вершине приписаны метки в согласие с некоторой системой типов.

Система типов определяет синтаксис типовых меток и правила, по которым их можно приписывать. Синтаксис обычно описывается в классических нотациях а ля BNF, а правила в виде типовых дробей. Например, так выглядят дроби для просто-типизированного λ-исчисления:

$$\frac{(x : \sigma) \in \Gamma}{\Gamma \vdash x : \sigma}\ ctx \qquad \frac{\Gamma \vdash M : \sigma \to \tau \quad \Gamma \vdash N : \sigma}{\Gamma \vdash M\ N : \tau}\ elim \to \qquad \frac{\{x : \sigma\} \cup \Gamma \vdash M : \tau}{\Gamma \vdash \lambda x^{\sigma}.\, M : \sigma \to \tau}\ intro \to$$

Типовые метки имеют чисто-синтаксическую природу, однако их можно проинтерпретировать. Самая популярная интерпретация — воспринимать типовую метку как множество. Так, метке $int \to int$ можно поставить в соответствие множество функций между множествами целых чисел $\mathbb{Z} \to \mathbb{Z}$.

1.3 Функции в Haskell

В своей основе Haskell представляет собой расширенное типизированное λ-исчисление, дополненное примитивными типами, возможностью декларировать новые имена, структурами данных и классами типов.

Примеры λ-абстракций в REPL окружении GHCi:

```
ghci> (\x -> x + 1) 4
5
```

Можно узнать тип функции в интерпретаторе (в реальности числа полиморфные, но об этом далее):

```
ghci> :t \x -> x + 1
\x -> x + 1 :: Int -> Int
```

Функциям можно давать имена. Именам можно приписывать типы, это рекомендуется делать явно для деклараций на верхнем уровне файлов исходного кода:

```
f :: Int -> Int
f x = x + 1
```

Если имя типа начинается с маленькой буквы, то это не конкретный тип, а типовая переменная, способная принимать различные значения в зависимости от места вызова. Такая возможность называется **параметрическим полиморфизмом**. Так, функция, которая просто возвращает свой аргумент, никак не ограничивает тип аргумента. Но в то же время тип результата должен совпадать с типом аргумента:

```
id :: a -> a
id x = x

ghci> :t id 5
id 5 :: Int
```

Функции могут принимать другие функции в качестве аргументов (такие функции называются **функциями высших порядков (higher-order functions)**. Имя функции может состоять из специальных символов, тогда она считается оператором и может применяться к своим операндам в инфиксном стиле:

```
($) :: (a -> b) -> a -> b
f $ x = f x
```

Пример рекурсивной функции, использующей охранные выражения для отличения базового случая рекурсии:

```haskell
factorial :: Int -> Int
factorial n
  | n < 1 = 1
  | otherwise = n * factorial (n - 1)
```

Упражнение 1 *Что выведет запрос* `ghci> :t uncurry (flip const)`*?*

Упражнение 2 *Что выведет запрос* `ghci> :t first . first` *при*

```haskell
first :: (a -> a') -> (a, b) -> (a', b)
```

Упражнение 3 *Реализуйте факториал с помощью техники аккумулирующего параметра.*

1.4 Данные в Haskell

В Haskell есть встроенная возможность объявлять новые типы данных на основании других типов, а так же создавать их экземпляры.

Зададим тип данных, описывающий животных:

```haskell
data Animal
  = Cat String Int
  | Dog String
```

Мы задали тип данных `Animal` и два способа создать значения этого типа: для кошек и собак. `Cat` и `Dog` — это **конструкторы данных**. Они представляют собой функции, реализация которых находится на стороне языка. Они выделяют память под экземпляры данного типа и позиционно размещают компоненты. Кошек мы описываем именем и оставшимся количеством жизней, а собак — только именем.

```haskell
Cat :: String -> Int -> Animal
Dog :: String -> Animal
```

Чтобы воспользоваться информацией, сохранённой в структуре данных, требуется деконструировать её с помощью паттерн-матчинга. Мы сопоставляем значение типа с образцом. Если образец похож на то, как было сконструировано значение, то он выбирается среди других образцов и переменные, задекларированные в нём, начинают ссылаться на соответствующее позиционно содержимое структуры данных:

```haskell
show :: Animal -> String
show animal = case animal of
  Cat name nLifes -> "This is cat " ++ name ++ " " ++ show nLifes
  Dog name -> "This is dog " ++ name
```

В Haskell есть специальный синтаксис для объявления полей с именованными метками.

```haskell
data Penguin = Penguin { getName :: String, getAge :: Int }
penguin = Penguin { getName = "Andrey", getAge = 500 }
```

Haskell генерирует функции-аксессоры для доступа к полям объекта:

```haskell
ghci> :t getName :: Penguin -> String
```

Часто функции в программировании частичные — при некоторых значениях аргументов они могут вернуть результат, а при некоторых — нет. Давайте моделировать это с помощью специального типа данных. Если есть вещественный результат, будем возвращать его. Если нет, будем возвращать специально выделенное константное значение этого типа.

```haskell
data MaybeD = NothingD | JustD Double
sqrt :: Double -> MaybeD
sqrt x = if x < 0 then NothingD else JustD (calcSqrt x)
```

Можно заметить, что так нам придётся объявлять по типу `MaybeT` для каждого типа `T`. Поэтому Haskell позволяет абстрагироваться в типе, аналогично тому как можно абстрагироваться по значениям в терме.

```haskell
data Maybe a = Nothing | Just a
sqrt :: Double -> Maybe Double
sqrt x = if x < 0 then Nothing else Just (calcSqrt x)
```

Заметьте, что сейчас `Maybe` — это не совсем тип, так как теперь нужно передать типовой параметр, чтобы получить конкретный тип. `Maybe` называют **типовым конструктором**.

Вместе с абстракцией на уровне типов появилась и аппликация типа к типу. А что если дать меньше параметров типовому конструктору, чем ожидается? А что если больше? Контроль за корректностью типовых аппликаций обеспечивает **система кайндов**[4]. Это простейшие "типы для типов", то есть синтаксические метки, контролирующие корректность записанных программистом типов. Так, обычные типы имеют метку (кайнд) `*`. Типовые конструкторы имеют стрелочные кайнды. Например, `Maybe :: * -> *`. Аппликация типового конструктора к типу подходящего кайнда убирает одну стрелку:

```haskell
ghci> :k Int
Int :: *
ghci> :k Maybe
Maybe :: * -> *
ghci> :k Maybe Int
Maybe Int :: *
```

Кроме совершенно новых типов данных, в Haskell можно объявлять типовые синонимы. Это имена, которые можно использовать вместо других типов, если, например, запись оригинального типа слишком длинная для повсеместного написания:

[4]Иногда в русскоязычной литературе кайнды называют родами типов, но мы не будем так говорить.

```haskell
type T a = VeryLongType Int (a -> AnotherLongType a)
```

Если тип данных содержит только один конструктор и только одно поле, то отсутствует необходимость в аллокации новой памяти, содержащей тег конструктора и набор ссылок на поля. В таком случае, в качестве значения такого типа можно всегда просто использовать значение оборачиваемого типа, оставляя новый тип присутствовать исключительно во время компиляции, снижая нагрузку во время исполнения. Для объявления таких типов-обёрток нужно воспользоваться ключевым словом `newtype` вместо `data`:

```haskell
newtype CourseId = CourseId Int64
newtype ModuleId = ModuleId Int64
```

Упражнение 4 *Определите кайнд конструктора типа*

```haskell
data Free f a = Pure a | Free (f (Free f a))
```

1.5 Классы типов в Haskell

Параметрический полиморфизм позволяет использовать один и тот же код для различных типов входных данных. Классы типов же позволяют одному идентификатору ссылаться на разные реализации для разных типов данных (что аналогично механизму перегрузки (overloading) в других языках). Классы типов, как говорят, являются механизмом **специального (ad-hoc) полиморфизма**. Так, мы можем задекларировать символ ==, выбор реализации которого зависит от выбора типа аргументов a:

```haskell
class Eq a where
  (==) :: a -> a -> Bool
```

Для каждого типа можно объявить свою собственную реализацию Eq:

```haskell
instance Eq CourseId where
  CourseId x == CourseId y = x == y

instance Eq a => Eq [a] where
  [] == [] = True
  x:xs == y:ys = x == y && xs == ys
```

Теперь в зависимости от конкретного типа a в месте вызова, будет выбрана подходящая реализация для этого типа:

```haskell
ghci> CourseId 1 == CourseId 2
False
ghci> [CourseId 1, CourseId 2] == [CourseId 1, CourseId 2]
True
```

Рассмотренные ранее параметрически-полиморфные функции ничего не могли делать со своими аргументами, кроме как возвращать их в качестве результата или передавать в другие полиморфные функции. Чтобы уметь делать что-то ещё, нужна какая-то дополнительная информация про тип, потому что иначе нет никакой гарантии, что над объектом данного типа можно делать все необходимые операции. Так, функция `suc n = n + 1` не будет работать для строчек, потому что для них, очевидно, не определена операция сложения. Поэтому некорректно будет приписать полиморфный тип `suc :: a -> a`.

Классы типов, в отличие от перегрузки, в том числе являются механизмом ограничения полиморфности функций. Мы можем явно задать, что функция требует не произвольный тип на вход, а произвольный тип, для которого определены обязательно нужные нам операции. Так, для типа `suc` достаточно ограничить тип условием наличия плюса для него (операция обозначаемая символом + объявлена в классе типов `Num`):

```
suc :: Num a => a -> a
```

Упражнение 5 *Реализуйте функцию, проверяющую равенство всех элементов данного списка.*

Упражнение 6 *Реализуйте инстанс полугруппы для функций.*

Упражнение 7 *Реализуйте проверку равенства функций.*

1.6 Монады в Haskell

Класс типов `Functor` объявляется для конструкторов типов и позволяет заменить в некотором контейнере все элементы одного типа на все элементы другого, оставляя структуру контейнера неизменной.

```
class Functor (f :: * -> *) where
  fmap :: (a -> b) -> f a -> f b

instance Functor [] where
  fmap :: (a -> b) -> [a] -> [b]
  fmap _ [] = []
  fmap f (x:xs) = f x : fmap f xs
```

В Haskell любая функция просто вычисляет результат некоторого типа. Однако в программирования часто требуются функции, которые не только вычисляют результат, но и делают что-то ещё. Например, изменяют какое-то состояние или пишут в консоль. Иными словами, производят побочные эффекты. В любом случае в Haskell мы можем только вернуть из функции только результат, поэтому такие побочные эффекты мы кодируем в качестве дополнительной структуры, оборачивающей чистый результат. Т.е. если функция без побочных эффектов возвращала какой-то тип `a`, то после добавления побочных эффектов в её реализацию, она будет возвращать некоторый тип *вычислений* `f a`.

- Если функция кидает ошибку, то `f = Maybe`.
- Если функция читает глобальное состояние типа `e`, то `f = e -> _`.
- Если функция читает глобальное состояние `s` и обновляет его, то `f = s -> (s, _)`.

Стандартная библиотека Haskell предоставляет несколько классов типов для работы со значениями вида `f a`. Они позволяют абстрагироваться от структуры `f` и работать со значениями `a` внутри, как будто нет никакой дополнительной структуры.

Первый такой класс типов позволяет писать выражения над вычислениями `f a`.

```haskell
class Functor f => Applicative (f :: * -> *) where
  pure :: a -> f a
  liftA2 :: (a -> b -> c) -> f a -> f b -> f c

instance Applicative Maybe where
  pure :: a -> Maybe a
  pure = Just

  liftA2 :: (a -> b -> c) -> Maybe a -> Maybe b -> Maybe c
  liftA2 _ Nothing _ = Nothing
  liftA2 _ _ Nothing = Nothing
  liftA2 f (Just x) (Just y) = Just (f x y)
```

Второй класс типов позволяет делать последовательную композицию вычислений в императивном стиле:

```haskell
class Applicative m => Monad (m :: * -> *) where
  (>>=) :: m a -> (a -> m b) -> m b

newtype State s a = State { runState :: s -> (s, a) }

instance Monad (State s) where
  (>>=) :: State s a -> (a -> State s b) -> State s b
  m >>= k = State \s ->
    let (s', x) = runState m s in
    runState (k x) s'
```

Теперь если мы определим базовые операции работы с состоянием, мы сможем писать код в императивном стиле с побочными эффектами.

```haskell
get :: State s s
get = State \s -> (s, s)

put :: s -> State s ()
put newS = State \oldS -> (newS, ())
```

```haskell
example :: State Int Int
example =
  get >>= \x ->
  put 42 >>= \() ->
  get >>= \y ->
  pure (x + y)
```

```
ghci> runState example 1
43
```

Для таких монадических цепочек существует специальный синтаксический сахар:

```haskell
example :: State Int Int
example = do
  x <- get
  put 42
  y <- get
  pure (x + y)
```

Упражнение 8 *Реализуйте `liftA3` через `liftA2`.*

Упражнение 9 *Реализуйте `>>=` через `join` и наоборот.*

Упражнение 10 *Два числа с консоли, поделите одно на другое нацело и распечатайте результат, если остаток не нулевой, распечатайте его тоже.*

2 Параметрический полиморфизм

Никакое нетривиальное свойство программ не может быть алгоритмически проверено[5]. Чтобы оставаться разрешимыми (в смысле проверки типов и/или вывода), многие системы типов жертвуют полнотой и, помимо некорректных программ, отвергают много корректных. В то же время системы типов также стараются предоставлять различные возможности, позволяющие протипизировать как можно больше корректных программ. Одна из них — параметрический полиморфизм.

Под **параметрическим полиморфизмом** мы будем подразумевать возможность кода единообразно работать с произвольными типами данных Strachey [2000], Cardelli and Wegner [1985], что позволяет во многих случаях избегать дублирования кода.

В этой главе мы рассмотрим, как описывают полиморфизм в самом простом виде — в типизированном λ-исчислении. Изучим различные формы параметрического полиморфизма и сопутствующие техники безопасного программирования. Проанализируем возможные способы эффективной реализации параметрического полиморфизма. И в завершение рассмотрим полиморфизм по рантайм-представлению, “полиморфизм по полиморфизму”.

2.1 Параметрический полиморфизм в языке

λ-абстракция позволяет обобщать выражения по значениям, каждая абстракция добавляет стрелку в тип выражения, аппликация же снимает стрелку:

$$\frac{x : \tau, \Gamma \vdash M : \sigma}{\Gamma \vdash \lambda x : \tau.\, M : \tau \to \sigma}\ Lam \quad \frac{\Gamma \vdash M : \tau \to \sigma \quad \Gamma \vdash N : \tau}{\Gamma \vdash M\ N : \sigma}\ App$$

В то же время Λ-абстракция позволяет обобщать выражения по типам, добавляя квантор в тип (Π-абстракцию), а применение терма к типу (**универсальная аппликация (universal application)**) позволяет выбрать, с каким конкретным типом этот терм планируется использовать [Pierce, 2002, глава 23]:

$$\frac{\Gamma \vdash M : \tau}{\Gamma \vdash \Lambda\alpha.\, M : \forall\alpha.\, \tau}\ TLam \quad \frac{\Gamma \vdash M : \forall\alpha.\, \tau}{\Gamma \vdash M\ \sigma : [\alpha \to \sigma]\ \tau}\ TApp$$

Таким образом, например, функция id фактически принимает *два аргумента: тип и значение*:

$$id : \forall\alpha.\, \alpha \to \alpha$$
$$id = \Lambda\alpha.\, \lambda x : \alpha.\, x$$
$$id\ nat : nat \to nat$$
$$id\ nat\ 42 : nat$$

В Haskell типовые абстракции и аппликации приписываются неявно механизмом вывода типов. Однако, есть расширения языка, которые позволяют их написать явно: TypeAbstractions, TypeApplications. Это может помочь, например, когда информации из терма не достаточно, чтобы вывести тип. Так, можно явно специализировать `id` на нужный тип:

[5] `https://en.wikipedia.org/wiki/Rice%27s_theorem`

```haskell
id :: forall a . a -> a
ghci> :t id @Int
id @Int :: Int -> Int
```

Кванторы также приписываются неявно в начале типа, следуя конвенции именования: конкретные типы начинаются с большой буквы, а полиморфные — с маленькой. Аналогично, у пользователя есть возможность явно приписывать `forall`'ы с помощью расширения ExplicitForAll. Это может понадобиться либо за тем, чтобы задать вручную порядок типовых абстракций, либо, чтобы иметь возможность сослаться на абстрагированный тип в теле функции (расширение ScopedTypeVariables).

Полиморфные типы данных задаются с помощью другой конструкции. Если ранее мы управляли типом с уровня термов универсальной аппликацией, то теперь мы хотим управлять типом на уровне типов. Для этого мы вводим λ абстракцию в типах, аппликацию в типах и, соответственно, β-редукцию. Система кайндов (пока) представляет собой простейшую "систему типов для типов" и обеспечивает well-formedness типов и строгую нормализуемость[6]. Например, мы можем написать тип пары, абстрагированный от конкретных типов компонент, чтобы пользователь мог выбрать нужные ему:

$$
\begin{aligned}
&Pair : * \to * \to * \\
&Pair = \lambda \tau^{*}\ \sigma^{*}. \forall \gamma. (\tau \to \sigma \to \gamma) \to \gamma \\
&pair : \forall \alpha\ \beta. \alpha \to \beta \to Pair\ \alpha\ \beta \\
&pair = \Lambda \alpha^{*}\ \beta^{*}. \lambda x^{\alpha}\ y^{\beta}. (\Lambda \gamma^{*}. \lambda f^{\alpha \to \beta \to \gamma}. f\ x\ y) \\
&fst : \forall \alpha\ \beta. Pair\ \alpha\ \beta \to \alpha \\
&fst = \Lambda \alpha^{*}\ \beta^{*}. \lambda p^{Pair\ \alpha\ \beta}. p\ \alpha\ (\mathbf{K}\ \alpha\ \beta)
\end{aligned}
$$

В Haskell вычислительную семантику полиморфных типов можно проследить в синонимах типов:

```haskell
type Pair a b = forall c . (a -> b -> c) -> c
intPair :: Pair Int Int -- forall c . (Int -> Int -> c) -> c
```

Обычные конструкторы типов номинативны. Например, `(Int, Int)` или `Maybe Int` никуда далее не вычисляются.

Haskell не позволяет создавать функции на типах по месту с помощью явной типовой лямбды[7] ввиду проблематичности этой конструкции для вывода типов. Однако именованные функции на типах есть, и мы рассмотрим их далее 3.2. В Scala существует нетривиальный трюк[8][9], который позволяет этого добиться. Scala3, однако, включила эту возможность непосредственно в язык[10].

[6] **Строгая нормализуемость** — любой порядок редукций приводит к нормальной форме.

[7] `https://stackoverflow.com/questions/4069840/lambda-for-type-expressions-in-haskell`

[8] (stackoverflow) Scala type lambdas.

[9] `https://stackoverflow.com/questions/9443004/what-does-the-operator-mean-in-scala`

[10] `https://docs.scala-lang.org/scala3/reference/new-types/type-lambdas.html`

2.1.1 Эмуляция типовых абстракций и аппликаций (Proxy)

В Haskell расширения, позволяющие вручную задавать типовые аппликации и абстракции появились сравнительно недавно[11]. До этого пользовались следующей техникой.

В стандартной библиотеке определён тип Proxy с одним параметром. Это **фантомный типовой параметр** — значения соответствующего типа не хранятся в структуре данных, он только позволяет размещать дополнительную информацию на уровне типов[12]. Соответственно, неинформативную константу Proxy можно проаннотировать нужным типом и передать в функцию, чтобы специализировать типовой параметр на нужный тип. Или можно принять Proxy и воспользоваться ScopedTypeVariables для типовых сигнатур в паттернах[13].

```
data Proxy a = Proxy

id :: Proxy a -> a -> a
ghci> :t id (Proxy :: Proxy Int)
id (Proxy :: Proxy Int) :: Int -> Int

id (Proxy :: Proxy a) x = (x :: a)
```

Иногда прокси-тип оставляют полиморфным, чтобы не зависеть от Data.Proxy. Вместо конкретного значения иногда передают специализированное значение $\bot$, а получатель, не зная тип, не сможет его форсировать (однако, любые вхождения $\bot$ в терм слишком настораживают, поэтому это скорее не очень хорошая практика).

```
id :: proxy a -> a -> a
id (_ :: proxy a) x = (x :: a)

ghci> :t id (undefined :: Proxy Int)
id (undefined :: Proxy Int) :: Int -> Int
```

2.1.2 First-class polymorphism

Существует возможность писать функции, которые принимают другие полиморфные функции в качестве аргументов. Типы таких функций называются **типами высшего ранга (higher-rank types)**, их можно использовать с расширением RankNTypes. Так, типовой параметр функции g определяет функция f, а не вызывающий функцию f:

```
f :: (forall a . a -> a) -> (Int, Char)
f g = (g @Int 42, g @Char 'a') -- универсальная аппликация для наглядности
ghci> f (\x -> x)
```

[11] TypeApplications, TypeAbstractions.

[12] https://wiki.haskell.org/Phantom_type

[13] Типовый параметр на самом деле имеет полиморфные кайнд `data Proxy (a :: k) = Proxy`, чтобы эта техника работала с типами произвольных кайндов (см. далее 2.1.5).

Проблема типов высшего ранга в том, что их вывод неразрешим, то есть глобальный вывод типов Haskell в этом случае перестаёт работать. Но если типы высшего ранга приписать вручную, остальной вывод будет работать как раньше. Например, числа Чёрча имеют высший ранг[14][15]:

```
suc :: (forall a . (a -> a) -> a -> a) -> (a -> a) -> a -> a
suc n s z = s (n s z)
```

Упражнение 11 *Какой ранг имеет тип* `Int -> (forall a . a -> a)`*?*

От многих проблем сопутствующих типам высших рангов можно избавиться, если создавать для них обёртки. Например, для чисел Чёрча можно создать обёртку `newtype Church`. Теперь код, работающий с обёрткой, может быть протипизирован типами первого ранга, только конструктор имеет тип высшего ранга:

```
newtype Church = Church (forall a . (a -> a) -> a -> a)
(+) :: Church -> Church -> Church -- rank 1
```

Аналогичный код можно написать и в Java (Kotlin):

```
interface Church { fun <a> fold(s: (a) -> a, z: a): a }
fun plus(n: Church, m: Church): Church = object : Church {
    override fun <a> fold(s: (a) -> a, z: a): a = n.fold(s, m.fold(s, z))
}
```

По умолчанию типовые параметры можно специализировать только на конкретные типы. Расширение ImpredicativeTypes позволяет специализировать типовые параметры на полиморфные типы (включающие `forall`'ы внутри себя) — **импредикативное применение**.

```
runST :: (forall s. ST s a) -> a
($) :: forall a b . (a -> b) -> a -> b
foo = runST $ ... -- типизируется только с ImpredicativeTypes
```

Higher-rank типы можно использовать как type-based escape analysis, иначе говоря, не позволять пользователю передавать некоторое значение вовне определённого скоупа. Так, например, Haskell предоставляет эффективную монаду `ST`, позволяющую в рамках ограниченного скоупа работать с мутабельными ячейками памяти Launchbury and Peyton Jones [1995][Maguire, a, 7.2, ST trick]:

```
newtype ST s a = ST (IO a)
runST :: (forall s. ST s a) -> a
```

[14] `https://okmij.org/ftp/tagless-final/course/Boehm-Berarducci.html`

[15] Любому интересующемуся языками программирования предлагается провести на сайте Олега Киселёва не один месяц жизни: `https://okmij.org/ftp/README.html`.

```haskell
sumTo :: Int -> Int
sumTo n = runST do
  ref <- newSTRef 0
  forM [0..n] \i -> modifySTRef ref (+ i)
  readSTRef ref
```

Заметим, что если попытаться вернуть из `runST` ссылку на мутабельную ячейку, то результирующий тип не пройдёт well-formedness проверку, так как будет содержать фантомный параметр `s`, который не будет нигде связан:

```haskell
newSTRef :: a -> ST s (Ref s a)
ghci> runST (newSTRef 0) :: Ref s Int -- ошибка
```

На практике, чтобы отличать такие локально связанные типовые переменные, используют концепцию уровней[16] Jones [2019].

Типы высших рангов вместе с импредикативным применением образуют **полиморфизм первого класса (first-class polymorphism)**, когда полиморфные типы могут использоваться почти так же свободно, как и любые другие. Классический алгоритм глобального вывода Хиндли-Милнера не справляется (и в общем случае задача неразрешима), так что существует большое количество решений, делающих различные компромиссы. Можно сделать вывод типов локальным, опирающемся только на соседние ноды AST и вспомогательные типовые аннотации Pierce and Turner [2000], Christiansen [2013], Dunfield and Krishnaswami [2019]. Либо же можно попытаться помочь глобальному выводу дополнительной предобработкой (Quick Look[17] Serrano et al. [2020], реализованный в Haskell с недавнего времени) или дополнительными регулирующими конструкциями (FreezeML Emrich et al. [2020]).

2.1.3 Higher-order/kinded polymorphism

Haskell позволяет также абстрагироваться по типам произвольных кайндов, а не только `Type`, как в `data` декларациях (**higher-order/kinded types (HKT)**[18]), так и в полиморфных функциях. Далее мы встретим немало примеров. Так, `Fix` имеет кайнд `(Type -> Type) -> Type`, а катаморфизм абстрагирован по типу стрелочного кайнда:

```haskell
newtype Fix f = Fix (f (Fix f))
cata :: forall (f :: Type -> Type) a . Functor f => (f a -> a) -> Fix f -> a
```

Далее мы рассмотрим технику, позволяющую типы высших порядков закодировать в языке, их не поддерживающем (см. 3.1.8).

[16] `https://okmij.org/ftp/ML/generalization.html`
[17] (youtube) A Quick Look at Impredicativity (Simon Peyton Jones)
[18] `https://serokell.io/blog/kinds-and-hkts-in-haskell`

2.1.4 Обобщённые алгебраические типы данных (GADTs)

Обобщённые алгебраические типы данных (generalized algebraic data types, GADTs) позволяют приписывать данным на уровне типов больше информации. В качестве модельного примера возьмём синтаксис крошечного языка программирования. Зададимся целью не допустить возможности конструирования в Haskell некорректных с точки зрения типов синтаксических деревьев.

```
data Expr = Const Int | IsZero Expr | If Expr Expr Expr
```

Как мы знаем, конструкторы данных в Haskell — это обычные функции с той лишь разницей, что их реализация генерируется компилятором (аллокация памяти, размещение полей...). У функций есть тип. Например, `IsZero :: Expr -> Expr`.

В Haskell есть синтаксис определения `data` через задание типов конструкторов[19]. Он совершенно аналогичен рассмотренному ранее, только гораздо более удобен для сложно организованных структур данных. Рассмотренный ранее тип термов `Expr` будет выглядеть следующим образом:

```
data Expr where
  Const :: Int -> Expr
  IsZero :: Expr -> Expr
  If :: Expr -> Expr -> Expr -> Expr
```

Для полиморфных структур данных, на примере списка, используется следующий синтаксис. Имя `elem` нужно исключительно для документации и больше никак его использовать нельзя, оно только маркирует наличие типового параметра и позволяет ему вручную задать кайнд[20].

```
data List (elem :: Type) where
  Nil :: List a
  Cons :: a -> List a -> List a
```

Добавим к `Expr` фантомный типовой параметр `ty`, обозначающий тип Haskell, в который должно быть проинтерпретировано данное выражение, и с помощью GADT зададим конкретные значения `ty` результирующим типам конструкторов. Так, мы говорим, что программа сконструированная с помощью `Const` вычисляется в число, `IsZero` вычисляется в булево значение, а условное выражение — в тип веток:

```
data Expr ty where
  Const :: Int -> Expr Int
  IsZero :: Expr Int -> Expr Bool
  If :: forall ty . Expr Bool -> Expr ty -> Expr ty -> Expr ty

eval :: Expr ty -> ty
```

[19] https://downloads.haskell.org/ghc/latest/docs/users_guide/exts/gadt_syntax.html#gadt-style

[20] Кайд можно не писать. Либо можно не писать имена параметров и просто приписать кайнд типовому конструктору: `data List :: Type -> Type where ...`.

Теперь мы можем написать безопасный типизированный интерпретатор. Обратите внимание, что при сопоставлении с образцами конструкторов, у нас уточняется информация о типовом параметре:[21]

```
eval :: Expr ty -> ty
eval = \case
  Const x  -> x           -- ty ~ Int
  IsZero t -> eval t == 0 -- ty ~ Bool
  If c t e -> if eval c then eval t else eval e
```

Далее мы рассмотрим как GADT в Haskell выражаются через более базовые механизмы языка 3.4.4.

2.1.5 Структуры на уровне типов, data promotion

Чтобы обрести больший контроль корректности программ, научимся кодировать произвольные структуры данных на уровне типов. В качестве модельной задачи зададим структуру данных, моделирующую вектор, но с контролем длины.

Для начала определим натуральные числа на уровне типов в стиле Пеано:

```
data Zero
data Suc n
```

Упражнение 12 *Сколько обитателей типа* `Suc (Suc Zero)`*?*

Теперь мы можем задать тип вектора, содержащий информацию о длине:

```
data Vec (size :: Type) (elem :: Type) where
  VNil :: Vec Zero a
  VCons :: a -> Vec n a -> Vec (Suc n) a

example :: Vec (Suc (Suc Zero)) Int
example = VCons 1 (VCons 2 VNil)
```

Для такого типа, например, можно написать безопасную функцию `zip`, работающую только на векторах одинаковой длины:

```
vzip :: Vec n a -> Vec n b -> Vec n (a, b)
vzip VNil VNil = VNil                                    -- n ~ Zero
vzip (VCons x xs) (VCons y ys) = VCons (x, y) (vzip xs ys) -- n ~ Suc n'
```

Заметьте, что в остальных ветках `vzip` должны возникнуть эквивалентности, начинающиеся с различных конструкторов, например, `Zero ~ Suc n`. Поскольку невозможно построить такие аргументы функции, Haskell позволяет соответствующие ветки не рассматривать.

[21] Тут используется удобное расширение LambdaCase, позволяющее не вводить лишние имена.

Упражнение 13 *Напишите функцию добавления в конец элемента вектора. Двигайтесь последовательно, заполняя типовые дыры и отслеживая возникающие эквивалентности.*

Сейчас язык кайндов, который должен контролировать типы, слишком беден. Действительно, кайнд `Suc` — `Suc :: Type -> Type`, соответственно ничто не мешает написать `Suc (Maybe Int)` В то же время он слишком ограничивающий, поскольку не поддерживает полиморфизм, что дало начало большому количеству дублирований а ля `Typeable (ty :: Type)`, `Typeable1 (ty :: Type`
Современный Haskell имеет расширение TypeData, позволяющее объявлять новые типы и кайнды подобно тому, как `data` позволяет объявлять новые типы.

```
type data Nat = Zero | Suc Nat
```

Теперь вектору можно приписать более точный кайнд:

```
data Vec (size :: Nat) (elem :: Type) where
  VNil :: Vec Zero a
  VCons :: a -> Vec n a -> Vec (Suc n) a
```

Упражнение 14 *Что выведет* `ghci> :k Vec`*?*

Другим вариантом добиться того же самого является использование DataKinds Yorgey et al. [2012]. Это расширение автоматически продвигает (promotion) все `data` декларации на уровень выше. А именно: любой конструктор типа также становится кайндом, а конструктор данных — конструктором типа. Так, в примере с числами, мы можем задекларировать натуральные числа как обычно и использовать на уровне типов:

```
data Nat = Zero | Suc Nat
ghci> :k Suc :: Nat -> Nat -- тут понятно что Suc используется как тип
```

Поскольку типы и термы в Haskell живут в разных пространствах имён, можно называть конструкторы типов и данных одинаково. Однако если продвинуть такой тип данных, возникнет неоднозначность: мы имеем в виду тип или продвинутый конструктор. Haskell позволяет указать явно, что речь идёт о продвинутом конструкторе с помощью одинарной кавычки.

```
data T = T Nat
ghci> :k T
T :: Type      -- про конструктор типа
ghci> :k 'T
'T :: Nat -> T -- про продвинутый конструктор данных
```

Не любые `data` декларации подходят для продвижения, в то же время `type data` декларации позволяют явно запросить структуру уровня типов и получить внятные ошибки, если декларация написана неправильно.
В случае продвижения полиморфного типа, мы получаем полиморфные кайнды (PolyKinds):

Term	Type	Kind
`Zero`	`Nat`	`Type`
`[Zero, Suc Zero]`	`[Nat]`	`Type`
`[]`	`forall a. [a]`	`Type`
`(:)`	`forall a. a -> [a] -> [a]`	`Type`
	`'Suc 'Zero`	`Nat`
	`'['Zero, 'Suc 'Zero]`	`[Nat]`
	`'[Int, Double]`	`[Type]`
	`'[]`	`forall k. [k]`
	`'(:)`	`forall k. k -> [k] -> [k]`

Рис. 5: Пример продвижений в Haskell.

```
data [a] = [] | (:) a [a]
ghci> :k '(:)
'(:) :: forall k . k -> [k] -> [k]
```

Примеры продвижения различных конструкций можно увидеть в таблице 5.
В качестве примера, зададим гетерогенный список, индексированный типами элементов:

```
data HList (tys :: [Type]) where
  HNil :: HList '[]
  HCons :: ty -> HList tys -> HList (ty ': tys)

example :: HList '[Int, Bool, Double]
example = HCons 42 $ HCons True $ HCons 12.5 HNil
```

Структуры данных тоже могут быть полиморфными по кайндам. Рассмотрим тип `Tagged`, позволяющий дополнить тип значения дополнительным типовым тегом. Кайнд тега может быть произвольным, поэтому, например, можем использовать встроенные в систему типов константы TypeLits (другой пример использования полиморфных кайндов мы видели ранее 2.1.1):

```
newtype Tagged (tag :: k) (a :: Type) = Tagged a
ghci> :t Tagged
Tagged :: forall k (tag :: k) a. a -> Tagged tag a

example :: Tagged ("dbId" :: Symbol) Int
example = Tagged 42
```

Современный Haskell в итоге пришёл к тому, что система типов не делает различий между типами и кайндами (рис. 6). В частности, `Type :: Type`. Это нужно для расширения возможностей Haskell в сторону программирования с зависимыми типами путём добавления несин-

таксических эквивалентностей для кайндов (TypeInType). *System FC* была представлена в работе Weirich et al. [2013][22][23].

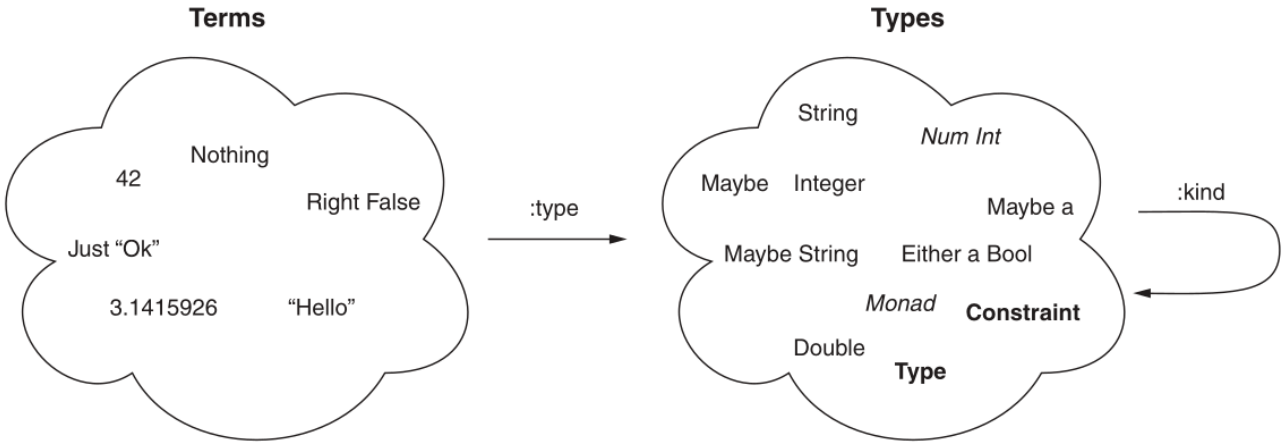


Рис. 6: Типы и кайнды — одно (Bragilevsky).

2.2 Реализация параметрического полиморфизма

Конвенция вызова[24] представляет собой набор соглашений между тем как функция компилируется и как должна вызываться. Например, функция принимает два аргумента, каждый размером в машинное слово, и возвращает один результат размером в машинное слово. Тогда сгенерированный низкоуровневый код этой функции может, например, ожидать, что оба аргумента передаются через специальную пару регистров, а складывать результат он будет в третий. В таком случае вызывающий код обязан предоставить аргументы в правильных регистрах и ожидать результата в некотором третьем, заранее оговоренном регистре.

В общем случае, конвенция вызова функции зависит от типов аргументов и результата. Нужно знать как минимум их размер, чтобы понять, размещать их в регистрах или на стеке. Нужно знать, это указатель (**reference type**) или значение само по себе (**value type**), чтобы понимать, как с ним работать. В структурах данных нужно знать смещения полей.

Таким образом, реализация параметрического полиморфизма в языке — это не тривиальная задача. Разные языки используют различные подходы, все со своими достоинствами и недостатками.

2.2.1 Мономорфизация

Мономорфизация — самый прямолинейный подход, компилируем полиморфные функции и структуры для каждого набора типовых аргументов. Так, если различных наборов типовых

[22](youtube) Мини-курс на русском языке про развитие Haskell в сторону зависимой типизации.
[23](youtube) Мини-курс на русском языке — система вывода типов Haskell.
[24]`https://en.wikipedia.org/wiki/Calling_convention`

аргументов, с которыми эта функция вызывается, например, 100 (что запросто может быть), то её код будет компилироваться сто раз и занимать в бинарнике в сто раз больше места. Так делают, например, C++ и Rust.

На самом деле всё ещё хуже. Если проект многомодульный и состоит из множества единиц компиляции (кусков, которые компилируются отдельно), то одна и та же специализация функции на типовые аргументы будет компилироваться заново во всех единицах компиляции, где такая специализация нужна. А затем, линкер будет заниматься удалением дубликатов, что тоже не самый быстрый и эффективный процесс.

+ Порождаемый код максимально эффективен для каждого типа;
+ Легко на этапе компиляции отрабатывают `is`-проверки значений на принадлежность определённому типу (в остальных подходах с этим всё сложно);
− Время компиляции крайне велико;
− Существенно увеличивается размер результирующего бинарного файла, что может быть критично для некоторых приложений;
− Может неэффективно работать из-за засорения кеша кода в процессоре;
− В интерфейсах не может быть полиморфных методов, так как мы не знаем в месте вызова, к какому именно наследнику относится вызываемый метод, и какой код нужно специализировать (аналогично, не работает higher-rank полиморфзм);
− К полиморфным функциям нельзя динамически линковаться (у них нет кода до специализации);
− В общем случае нельзя поддержать variance, потому что код компилируется для конкретного типа и в общем случае не может работать для произвольного подтипа или супертипа (если reference и value типы могут находиться в одной иерархии подтипизации).

Некоторые языки (например, C++ и Zig) не делают инстанциацию скрытой деталью реализации языка, а предоставляют её как инструмент пользователям. Это даёт следующие возможности:

- Если разрешить использовать значения в типах, инстанциация может использоваться как механизм вычислений на этапе компиляции.
- Если отложить проверку ошибок на стадию инстанциирования, то мы получим своего рода статическую утиную типизацию. Это позволит не описывать сложные сигнатуры полиморфных функций. Однако тогда функции для тестирования придётся вручную инстанциировать против всевозможных типов, иначе нельзя понять статически, компилируется ли она хотя бы против этих типов или нет.

2.2.2 Стирание типа

Можно всё сделать наоборот, унифицировав значения, которые приходят на вход полиморфным функциям и хранятся в полиморфных структурах данных, вместо того, чтобы компилировать код под каждый тип.

Пусть каждое значение будет аллоцировано в куче и передаваться по указателю. Тогда мы сможем переиспользовать один и тот же код для разных типовых аргументов — он просто будет ожидать указатели.

+ Каждая функция компилируется ровно один раз — быстро;
+ Можно динамически загружать новые полиморфные функции и типы и использовать их друг с другом;
+ Гибкость — вариантность, полиморфные методы в интерфейсах, higher-rank types и т.д. просто работают;
− Аллокация в куче и разыменование указателя может очень сильно замедлить код;
− Поскольку информация о типах стирается, нельзя ничего сделать с типовым аргументом, не имея его обитателей (например, запросить рефлексией информацию или сделать `is` проверку).

Такого подхода придерживаются JVM, Haskell и, как правило, другие функциональные языки ввиду его гибкости и скорости компиляции.

Особую проблему вызывает работа с примитивами и другими value-типами, потому что каждое значение приходится сначала боксить (переносить в кучу), а потом уже использовать в полиморфном контексте. Поэтому языки борются с этим как могут. Некоторые языки урезают диапазоны значений примитивов, чтобы зарезервировать бит, определяющий, это указатель или значение. Код консультируется с этим битом для работы (похоже на 2.2.4). Так делают, например, OCaml и Koka. Агрессивный инлайнинг вместе с другими оптимизациями тоже помогают Blanchet [1999]. Java, например, также пытается аккуратно двигаться в сторону возможности мономорфизации[25][26].

2.2.3 Гибридный подход

C# реализует гибридный подход[27]. Они различают значения, хранимые в куче — reference types, и значения, хранимые на стеке — value types. Для первых они генерируют одну специализацию, работающую с указателями. Для каждого набора value-типов они генерируют лениво, в рантайме, специализации.

То есть следы дженериков в таком подходе есть и промежуточном представлении CIL, и в рантайме.

+ value-типы хранятся и передаются as-is без боксинга;
+ Доступна рефлексия по дженерикам;
+ Небольшое время компиляции;
− Инстанциация в рантайме замедляет исполнение;
− Variance работает только для reference types (что странно — есть “правильная” подтипизация, а есть “неправильная”).

[25] Type Specialization of Java Generics - What If Casts Have Teeth?
[26] `https://cr.openjdk.org/~jrose/values/parametric-vm.html`
[27] Generics in the runtime (C# programming guide).

```
struct value_witness_table {
  size_t size, align;
  void (*copy_init)(opaque *dst, const opaque *src, type *T);
  void (*copy_assign)(opaque *dst, const opaque *src, type *T);
  void (*move_init)(opaque *dst, opaque *src, type *T);
  void (*move_assign)(opaque *dst, opaque *src, type *T);
  void (*destroy)(opaque *val, type *T);
};
```

Рис. 7: Swift value witness table.

Example:

```
func f<T>(_ t: T) -> T {
  let copy = t
  return copy
}
```

Implementation:

```
void f(opaque *result, opaque *t, type *T) {
  opaque *copy = alloca(T->vwt->size);
  T->vwt->copy_init(copy, t, T);
  T->vwt->move_init(result, copy, T);
  T->vwt->destroy(t, T);
}
```

Рис. 8: Код полиморфной функции, порождаемый компилятором Swift.

2.2.4 Использование виртуальной таблицы свойств типов

Swift[28] вместе с каждым типовым параметром передаёт value witness table (рис. 7). Это таблица со всей необходимой информацией о типе: размер и выравнивание, что нужно сделать при копировании и перемещении объекта (например, инкрементировать счётчик ссылок). Таким образом, скомпилированный код постоянно обращается к этой таблице и делает виртуальные вызовы функций из неё (рис. 8).

- \+ Небольшое время компиляции;
- \+ Предсказуемая эффективность (не приводит к неожиданным паузам в рантайме);
- \+ Эффективная работа с value-значениями;
- \+ Высокая гибкость;
- \+ Информация о типах не стирается;
- − Серьёзный константный оверхед на динамические вызовы через таблицу, эффективность очень сильно зависит от компиляторных оптимизаций.

Своего рода реализация параметрического полиморфизма через специальный.

[28](youtube) 2017 LLVM Developers' Meeting: "Implementing Swift Generics"

2.3 Полиморфизм по конвенции вызова

Как мы уже обсуждали выше 2.2.2, параметрический полиморфизм в Haskell реализуется следующим образом: все значения хранятся в куче и передаются в полиморфные функции по указателю. Однако, если для вычислительного кода важна производительность, такой подход не годится ввиду большой нагрузки на подсистему управления памятью и множества индирекций. Поэтому Haskell позволяет также писать код с использованием unboxed значений. А если конвенция вызова не принципиальна, можно по ней абстрагироваться и писать один код для boxed и unboxed значений Eisenberg and Peyton Jones [2017].

2.3.1 Разновидности runtime представлений в Haskell

	Boxed	Unboxed
Lifted	*Int* *Bool*	
Unlifted	$ByteArray_{\#}$	$Int_{\#}$ $Char_{\#}$

Рис. 9: Виды значений в Haskell с примерами Eisenberg and Peyton Jones [2017].

На рисунке 9 можно увидеть классификацию значений в Haskell с примерами типов. **Unboxed типы** — их значения удерживаются и передаются по значению. **Boxed**, соответственно, наоборот, передаются по указателю и хранятся в куче. Обычный `Int` является просто декларацией следующего вида, где `I#` — это обычный конструктор с необычным именем, содержащий unboxed значение.

```
1 data Int = I# Int#
```

Lifted типы — содержат $\perp$ в качестве значения. Иначе говоря, могут содержать отложенные вычисления (это для них специальным образом обеспечивают компилятор и рантайм). **Unlifted типы** — наоборот, не могут быть отложенными. Операции, производящие значения unlifted типов всегда энергичные. Свойство lifted/unlifted называют **levity**. Чтобы распространить дальнейшее изложение на энергичные языки, можно levity заменить на boxity и всё останется справедливым.

`#` в именах типов и функций — это конвенция, показывающая, что где-то рядом происходит работа с unlifted значениями[29].

Также в Haskell есть unboxed кортежи, которых не существует на этапе исполнения. Например, следующая функция как бы возвращает пару значений, но в действительности компилятор может их разместить, например, в паре регистров. Соответственно, паттерн-матчинг по таким кортежам, просто позволяет сослаться на каждое из этих значений.

[29]Нужно подключить расширение MagicHash, чтобы пользоваться # в идентификаторах.

```haskell
divMod# :: Int -> Int -> (# Int, Int #)
case divMod# n k of (# quot, rem #) -> ...
```

Соответственно, нет никакого различия между по-разному вложенными unboxed кортежами:

```haskell
(# A, (# B, C #)) ≡ (# #( A, B #), C #) ≡ (# A, B, C #)
```

2.3.2 Классификация значений по runtime представлению

Значения различных типов могут быть на этапе исполнения устроены по-разному. То есть нам нужна некоторая система классификации типов. Но такая система в Haskell уж есть — кайнды. Опишем в виде структур данных предметную область, а потом продвинем на нужный уровень с помощью DataKinds 2.1.5.

Стандартная библиотека Haskell предоставляет следующие типы данных:

```haskell
TYPE :: RuntimeRep -> Type

data Levity = Lifted | Unlifted

data RuntimeRep = BoxedRep Levity
                | IntRep | DoubleRep
                | TupleRep [RuntimeRep]
                | SumRep [RuntimeRep]
                | ...

type LiftedRep = BoxedRep Lifted

type Type = TYPE LiftedRep
```

`TYPE` — это магический тип, определённый в компиляторе. Он параметризован runtime-представлением значений. Теперь привычный `Type` — это частный случай с boxed lifted значениями.

- `Int :: TYPE (BoxedRep Lifted)` или `:: Type`
- `IntRep` и `DoubleRep` соответствуют представлению численных констант (в зависимости от архитектуры процессора, целые числа и числа с плавающей запятой может быть необходимо располагать в различных специальных регистрах)
  `Int# :: TYPE IntRep`
- `Maybe Int :: Type`
- `Maybe :: Type -> Type`
- `TupleRep` и `SumRep` — unboxed алгебраические типы, представления параметризованы представлениями хранимых значений
  `(# Int, Bool #) :: TYPE (TupleRep '[LiftedRep, LiftedRep])`
- Для простоты, типы вложенных кортежей не унифицируются

```
(# Int#, (# Int, Double# #) #)
  :: TYPE (TupleRep '[IntRep, TupleRep '[LiftedRep, DoubleRep]])
```

2.3.3 Representation polymorphism

Выставив runtime-представление в структуре кайндов, мы теперь можем параметризоваться по ним. Например, кайнд функциональной стрелки выглядит следующим образом[30]:

```
ghci> :k (->)
(->) :: forall {q :: RuntimeRep} {r :: RuntimeRep}. TYPE q -> TYPE r -> Type
```

Упражнение 15 *Подумайте, почему функция имеет boxed тип. Может ли быть иначе? Может ли это быть полезным?*

К сожалению, Haskell выставляет довольно строгое ограничение: связыватели не могут иметь тип, полиморфный по runtime представлению. Можно легко предположить, почему, — нельзя сгенерировать код функции для работы с параметром произвольного рантайм-представления. Это можно решить только мономорфизацией 2.2.1, но Haskell избегает этого подхода[31]. Сообщество также пытается найти другие решения[32] (что-то вроде 2.2.4).

Например, изначально оператор аппликации был обобщён только по возвращаемому типу. Это не порождает проблем, так как вызывающий код сможет вывести представление и сгенерировать подходящий код:

```
($) :: forall r a (b :: TYPE r). (a -> b) -> a -> b
f $ x = f x
```

Однако, было замечено, что для оператора аппликации можно получить другую реализацию, не использующую levity-полиморфное связывание[33]:

```
($) :: forall ra rb (a :: TYPE ra) (b :: TYPE rb). (a -> b) -> a -> b
($) f = f
```

Таким образом, в Haskell полиморфизм по представлениям несколько вырожден и помогает лишь в небольшом количестве случаев, однако немаловажных. Если позволить мономорфизацию по `RuntimeRep` параметрам, получится система аналогичная гибридной реализации параметрического полиморфизма 2.2.3, только с большим контролем со стороны программиста над мономорфизацией.

[30] Выключить упрощения: `ghci> :set -fprint-explicit-foralls -fprint-explicit-runtime-reps`
[31] https://gitlab.haskell.org/ghc/ghc/-/issues/14917
[32] https://mail.haskell.org/pipermail/haskell-cafe/2023-January/135770.html
[33] https://gitlab.haskell.org/ghc/ghc/-/merge_requests/10131

3 Специальный (ad-hoc) полиморфизм

Как-то Joe Fasel в разговоре с Philip Wadler высказал идею того, что перегрузка функций (overloading) должна находить своё отражение в типах. Wadler понял его неправильно Hudak et al. [2007]. Но то, что он понял, — оказалось классами типов Wadler and Blott [1989].

Christopher Strachey ввёл классификацию полиморфизма на две категории Strachey [2000]. Параметрический — один и тот же код работает с данными различных типов. **Специальный (ad-hoc) полиморфизм** — код выбирается в зависимости от типа. Например, один и тот же символ умножения по-разному действует на целые числа и на числа с плавающей точкой.

Перегрузка в языках обозначает возможность назвать несколько функций с различными наборами входных параметров одинаково. В месте вызова компилятор статически определяет по типам аргументов, какую из них действительно следует вызвать.

```
string toString(x: int) { ... }
string toString(fmt: String, d: double) { ... }
```

Классы типов обязуют сначала задекларировать именованную сущность (собственно, класс типов), включающую в себя пачку деклараций функций, которые могут быть перегружены для различных типов.

```
class Show a where
  show :: a -> String

instance Show Int where
  show :: Int -> String
  show = ...
```

Необходимо заметить, что декларация класса типов содержит формальный типовой параметр, по вхождениям которого в тип функции, собственно, выбирается перегрузка. Таких параметров может быть много (MultiParamTypeClasses), они могут иметь стрелочные кайнды. Например, в случае класса типов `Applicative`, выбор реализации операции `pure` будет происходить по типовому конструктору результата, то есть даже не по полноценному типу.

```
class Functor f => Applicative (f :: Type -> Type) where
  pure :: a -> f a
  ...

instance Applicative Maybe where
  pure :: a -> Maybe a
  ...
```

Также, в отличие от перегрузки, классы типов совместимы с параметрическим полиморфизмом. Так, в типе полиморфной функции нельзя указать, что для типа должна присутствовать определённая перегрузка. Классы типов же позволяют ограничить набор возможных типовых аргументов теми, для которых реализован инстанс нужного класса типов:

```haskell
showPrefixed :: Show a => a -> String -> String
```

Если сравнивать классы типов с переопределением (overriding) в ООП языках, то разрешение вызова виртуальной функции происходит с использованием таблицы, хранящейся объекте первого параметра (получателя вызова, receiver). Классы типов же опираются исключительно на тип, поэтому, например, возможно определение констант в классах типов[34]:

```haskell
class Enum a => Bounded a where
  minBound :: a
  maxBound :: a
```

В то же время, классы типов не являются типами, а, скорее, предикатами на типах. Тип удовлетворяет такому предикату, или свойству, если для него есть соответствующий инстанс. Поэтому, в частности, привычный способ в ООП создать гетерогенную коллекцию элементов, имеющих общий интерфейс, напрямую не сработает с классами типов. Например, такой тип не будет корректным: `[Show]`. Мы вернёмся к этой проблеме в 3.4.2.

3.1 Классы типов в языке

Несмотря на поразительное могущество, идея реализации классов типов крайне проста. Она была уже во всей полноте представлена в первой работе Wadler and Blott [1989]. В дальнейших работах уточнялся механизм вывода типов в виде сведения к классической системе типов в стиле Hindley-Milner Hall et al. [1996]. Остальные работы, в основном, предлагают огромное разнообразие различных расширений и приложений Jones et al. [1997].

3.1.1 Словари

Рассмотрим идею реализации классов типов на примере полиморфной сортировки. Сортировка для списка элементов конкретного типа пишется тривиально:

```haskell
sort :: [Int] -> [Int]
sort = \case [] -> []; x:xs -> insert x (sort xs)
  where
    insert x xs = let (l, r) = List.partition (<) x) xs in l ++ x : r)
```

В реализации единственная информация о типе, которой мы пользуемся — порядок на его обитателях. Таким образом, при переходе к полиморфной сортировке, нам нужно принять словарь с предикатами, задающими нужный порядок для данного типа.

```haskell
data OrdDict a = OrdDict { less :: a -> a -> Bool }

sort :: OrdDict a -> [a] -> [a]
```

[34] Современные ООП языки, тем не менее, стремятся поддержать статические функции в интерфейсах, что делает их ближе к классам типов и позволяет делать похожие вещи. Например, Swift.

```haskell
sort d@OrdDict{ less } = \case [] -> []; x:xs -> insert x (sort d xs)
  where
    insert x xs = let (l, r) = List.partition (`less` x) xs in l ++ x : r
```

Теперь, чтобы воспользоваться сортировкой на списке чисел, нужно сконструировать нужный рекорд и вызвать с ним функцию на списке конкретных типов:

```haskell
intOrd :: OrdDict Int
intOrd = OrdDict { less = (<) }
```

```
ghci> sort intOrd [3, 2, 1]
```

Возможна ситуация, когда инстанс для одного типа зависит от инстанса для другого Например, порядок на списках можно получить автоматически, зная порядок на элементах. В случае словарей мы это моделируем функцией между словарями:

```haskell
listDict :: OrdDict a -> OrdDict [a]
listDict d = OrdDict { less = ... less d ... }
```

Теперь мы можем сортировать список списков, конструируя нужный словарь:

```
ghci> sort (listDict intDict) [[3, 2], [2, 1], [0]]
```

Сравнение явной передачи словарей и классов типов можно увидеть в следующей таблице:

1. Определение словаря функций

```haskell
data MyOrd a = MyOrd
  { less :: a -> a -> Bool }
```

2. Экземпляр словаря для конкретного типа
   - Именованное значение

```haskell
intMyOrd :: MyOrd Int
intMyOrd = MyOrd { less = (<) }
```

3. Явный параметр функции

```haskell
sort :: MyOrd a -> [a] -> [a]
```

4. Передаётся пользователем

```haskell
test = sort intMyOrd [3, 2, 1]
```

1. Определение класса типов

```haskell
class MyOrd a where
  less :: a -> a -> Bool
```

2. Объявление типа представителем класса типов
   - Не имеет имени

```haskell
instance MyOrd Int where
  less = (<)
```

3. Неявный параметр функции

```haskell
sort :: MyOrd a => [a] -> [a]
```

4. Передаётся компилятором

```haskell
test = sort [3, 2, 1]
```

Таким образом, словарь — это **свидетель (witness)** или доказательство того, что тип удовлетворяет ограничению.

Упражнение 16 *Какой словарь будет соответствовать higher-kinded классу типов* `Functor`*?*

3.1.2 Неявные аргументы

Можно думать так, что слева от `=>` передаются неявные аргументы функций, выводимые компилятором из контекста. То есть, например, не стоит удивляться вхождениям `=>` в типе аргумента, это просто функция с неявным аргументом. Так, следующий код не скомпилируется, потому что в месте использования переменной `y` нет значения типа `Show b`:

```
f :: (Show b => b) -> b
f x = x -- ошибка
```

Можно это значение принять в функции `f`, тогда оно автоматически пропагируется в `y`:

```
f :: Show b => (Show b => b) -> b
f x = x
```

Расширение ImplicitParams даёт возможность делать некоторые аргументы функции неявными. Фактически, это реализация динамического связывания в статическом языке Lewis et al. [2000] (см. далее 5.3.1). Неявные аргументы берутся из скоупа по имени и подставляются автоматически:

```
sortBy :: (a -> a -> Bool) -> [a] -> [a]

sort :: (?cmp :: a -> a -> Bool) => [a] -> [a]
sort = sortBy ?cmp
```

Haskell также предоставляет возможность сохранять словари в структуры данных:

```
data ShowDict a where
  ShowDict :: Show a => ShowDict a

f :: ShowDict b -> (Show b => b) -> b
f d x = case d of ShowDict -> x -- в скоупе доступен инстанс Show b
```

Упражнение 17 *Возможна ли именно такая семантика в энергичном языке? Почему?*

3.1.3 Вывод инстансов

Чтобы вызвать ограниченно-полиморфную функцию, GHC производит вывод инстансов или, иначе говоря, автоматически конструирует свидетелей. Вывод инстансов тесно интегрирован с общей системой вывода типов Haskell Peyton Jones [2019].

В действительности вывод инстансов это не что иное, как *задача населения типа*. Действительно, после трансляции в Core (промежуточное представление в GHC), классы типов представляют собой словари функций. У нас в контексте имеются конкретные словари и функции, позволяющие из одних словарей получать другие. Требуется найти терм, конструирующий словарь нужного типа.

Пусть, например, внутри функции `f :: Show a => ..` происходит вызов ограниченно–полиморфной функции `g :: Show [a] -> ...` То есть, у нас имеется словарь `d1 :: ShowDict a`, а так же функция `d2 :: ShowDict a -> ShowDict [a]`, пришедшая из импортов[35]. Необходимо сконструировать терм типа `ShowDict [a]`. Очевидно, это будет просто аппликации одного к другому: `d2 d1`.

Вывод инстансов происходит рекурсивно. Чтобы вывести `ShowDict [a]`, выводится сначала посылка `ShowDict a`. То есть получается рекурсия по структуре типа. Иначе говоря, вывод инстансов можно эксплуатировать как вычислительный примитив уровня типов. Так, например, мы можем опускать информацию из типов в термы (аналогично GHC.TypeLits):

```
type data Nat = Zero | Suc Nat

class KnownNat (n :: Nat) where
  natVal :: Int

instance KnownNat Zero where
  natVal = 0

instance KnownNat n => KnownNat (Suc n) where
  natVal = 1 + natVal @n

ghci> natVal @(Suc (Suc Zero))
-- выведется natVal {knownSuc (knownSuc knownZero)}
```

В общем случае процесс населения типа, как можно предположить по вычислительной аналогии, неразрешим. Поэтому GHC накладывает большое количество ограничений на вид инстансов, которые гарантируют тотальность вывода. Подробно эти ограничения описаны в Sulzmann et al. [2007a]. Также GHC предоставляет различные расширения, ослабляющие эти ограничения и перекладывающие часть ответственности на плечи программиста[36]. Например, с UndecidableInstances можно легко написать разворот списка типов на этапе компиляции, как и любую другую функцию:

```
class Reverse (acc :: [Type]) (tys :: [Type]) where
  showReverse :: String

instance ShowT acc => Reverse acc '[] where
  showReverse = showTypes @acc

instance Reverse (ty : acc) tys => Reverse acc (ty : tys) where
  showReverse = showReverse @(ty : acc) @tys
```

[35] Инстансы можно импортировать пустым импортом: `import Module ()`.

[36] https://downloads.haskell.org/ghc/latest/docs/users_guide/exts/instances.html

```
ghci> showReverse @'[] @'[Char, Int, Double]
```

Можно заметить, что процесс вывода классов типов очень похож на вычисление логических программ, например, на Prolog, только без backtracking'а (перебора различных вариантов решений в поисках подходящего). Как, впрочем, и вывод типов в Haskell Peyton Jones [2019] в целом: собранные по программе эквивалентности можно рассматривать как логическую программу, решение этой системы типовых уравнений — как исполнение этой программы.

Между классами типов и выводом типов существует интересная синергия (рис. 10)[37]. Исходя из термов, выводятся типы. Затем, исходя из типов, выводятся инстансы классов типов. То есть мы пишем какой-то интересный интеллектуальный код, а параллельно с нами компилятор выписывает неинтересный код.

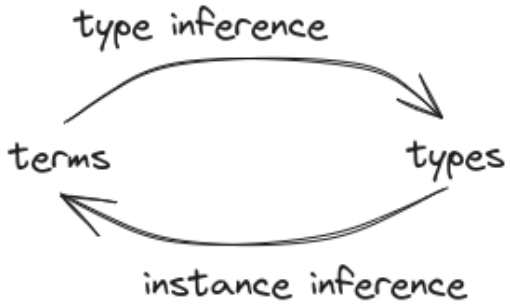


Рис. 10: Классы типов + вывод инстансов = кодогенерация.

Вывод инстансов опирается только на вид "головы" декларации — справа от `=>`, а ограничения слева применяются постфактум. Это можно использовать, чтобы писать более общие инстансы. Так, например, работает **constraint trick**[38], позволяющий резолвить ad-hoc полиморфные функции в параметрически-полиморфном контексте.

3.1.4 Построение типа по значению

После того как мы научились опускать значения из типов, закономерно научиться обратному — поднимать значения в типы. Воспользуемся техникой, описанной в Kiselyov and Shan [2004]. Существует соответствующая библиотека Data.Reflection (см. далее 3.4.7).

В действительности мы, конечно, не можем честно получить синтаксически тип нужного размера, просто потому, что типы существуют строго до стадии исполнения. Однако, как мы знаем, словари классов типов имеют воплощение в рантайме (случай полиморфной рекурсии — как раз пример, когда этого нельзя полностью избежать). Поэтому воспользуемся continuation passing style, который будет подробно рассмотрен далее в главе 6: вместо

[37] (youtube) Hackett: a metaprogrammable Haskell.
[38] `https://chrisdone.com/posts/haskell-constraint-trick/`

того, чтобы вернуть результат, примем продолжение, умеющее работать с любым типом с KnownNat (пользуемся типовыми абстракциями и аппликациями, см. 2.1)[39]:

```
reify :: Int -> (forall n. KnownNat n => a) -> a
reify n k
  | n <= 0 = k @Zero
  | otherwise = reify (n - 1) \@n' -> k @(Suc n')
```

Продолжение, передаваемое в рекурсивный вызов, захватывает словарь для типа n и конструирует словарь для Suc n.

Наконец, можем написать следующую удивительную тождественную функцию, поднимающую сначала значение в тип, а потом опускающее тип обратно в термы:

```
wonderId :: Int -> Int
wonderId n = reify n (\@t -> natVal @t)
```

3.1.5 Имплиситы и когерентность

Классы типов можно не делать специальным языковым механизмом, но вместо этого предоставлять на языковом уровне неявные параметры и население, достаточные для реализации классов типов.

Так, в Scala существует механизм имплиситов (implicits) Křikava et al. [2019][40]. Параметры функций могут быть помечены ключевым словом implicit, тогда Scala попытается их вывести самостоятельно с помощью доступных в скоупе implicit деклараций. Объявления переменных, функций и конструкторов объектов также могут быть помечены implicit, тогда они будут использоваться при населении. Теперь мы можем смоделировать словарь функций, например, с помощью интерфейсов (которые в Scala называются trait) и ООП синглтонов, чтобы получить классы типов Oliveira et al. [2010]:

```
// Пачка функций.
trait Show[T] {
    def show(x: T): String
}

// Обёртка для удобства вызова.
def show[T](x: T)(implicit ev: Show[T]): String = ev.show(x)

// Объект-синклтон, значение для пачки функций.
implicit object intShow extends Show[Int] {
    def show(x: Int): String = x.toString
}
```

[39] В не самых свежих версиях GHC потребуется воспользоваться техникой Proxy из 2.1.1.

[40] Дизайн неявных параметров в Scala3 изменился (youtube) Scala Implicits Revisited, Martin Odersky.

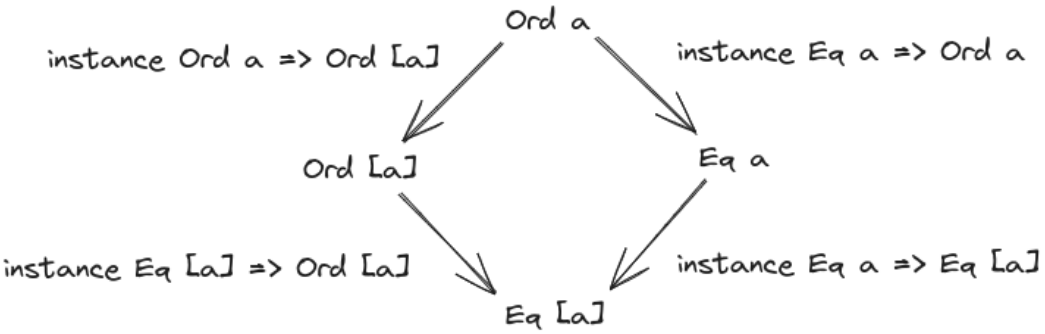


Рис. 11: Когерентность инстансов — диаграмма коммутирует.

```
11 def showAll[T](xs: List[T])(implicit ev: Show[T]): String =
12     xs.map(show(_)).join(", ")
```

Как мы говорили ранее (2.1), работа с типовыми параметрами похожа на работу с обычными. Также, вывод типов можно рассматривать как процесс восстановления пропущенных типовых аппликаций. Заметим, что примерно этим же занимается и механизм вывода имплиситов. Таким образом, при попытке сделать функции с имплиситами функциями первого класса, будут возникать сложности схожие со сложностями first-class полиморфизма (2.1.2).

В языках с зависимыми типами неявные параметры[41] особенно нужны, потому что, например, типы — это ровно такие же параметры функции, как и все остальные. Поэтому вывод типов — это фактически вывод неявных аргументов функций. Более того, зависимые функции, вместе с аргументами часто принимают доказательства каких-то свойств этих аргументов, которые тоже хочется по возможности выводить из контекста автоматически. Такой механизм вывода можно переиспользовать для эмулирования классов типов[42] Devriese and Piessens [2011]. В обратную сторону тоже работает — можно механизмы зависимой типизации эмулировать классами типов McBride [2002].

Как мы увидим далее, неявные параметры сами по себе тоже нужны как статическая аппроксимация динамических свободных переменных для реализации системы эффектов (см. далее **??**). Однако, иметь классы типов отдельной языковой возможностью всё же полезно, несмотря на то, что они, вроде бы, представляют собой те же неявные параметры (пусть и с рекурсивным механизмом населения). Так, можно поддержать важное свойство при соблюдении всех ограничений, т.е. при отсутствии **orphan instances**[43]. **Когерентность инстансов (coherence)** — для одного типа все инстансы данного класса типов, полученные разными способами, неотличимы (рис. 11). Соответственно, не имеет значения происхождение того или иного инстанса. Иначе говоря, об этом можно не думать, это снимает существенное

[41] https://agda.readthedocs.io/en/v2.7.0.1/language/implicit-arguments.html
[42] https://agda.readthedocs.io/en/v2.7.0.1/language/instance-arguments.html
[43] https://stackoverflow.com/questions/3079537/orphaned-instances-in-haskell

количество когнитивной нагрузки и упрощает рефакторинг[44]. В то время как остальные подходы требуют трепетного отношения к контексту вызова, потому что из него может прийти неожиданная реализация.

3.1.6 Правила (rules) и специализация

GHC позволяет прямо в коде, с помощью специально прагмы, указывать оптимизирующие правила переписывания для компилятора[45] Jones et al. [2001]. Например:

```
{-# RULES
  "map/map"    forall f g xs.  map f (map g xs) = map (f . g) xs
  "map/append" forall f xs ys. map f (xs ++ ys) = map f xs ++ map f ys
 #-}
```

Первый закон представляет собой не что иное, как закон функторов. В идеале, мы формулируем законы на этапе дизайна Maguire [b], проверяем их выполнение с помощью property-based testing[46], а потом используем их для оптимизаций.

Можно переписать полиморфную версию функции на специализированную, если типы подходят. Для этого нужно реализовать специализированную версию (совпадение семантики — полностью ответственность программиста) и задать соответствующее правило переписывания:

```
genericLookup :: Ord a => Table a b   -> a   -> b
intLookup     ::          Table Int b -> Int -> b

{-# RULES "genericLookup/Int" genericLookup = intLookup #-}
```

Основной эффект такой оптимизации — гарантированное превращение динамических вызовов функций классов типов в статические (потому что тип известен, следовательно, — и соответствующий ему словарь).

3.1.7 Отступление: дефункционализация

Дефункционализация (defunctionalization) — техника избавления от функций высших порядков в программе[47] Xia. Впервые предложена в Reynolds [1972, 1998].

Идея заключается в том, чтобы заменить каждое создание лямбда-функции вызовом конструктора некоторого алгебраического типа данных. А каждый call-cite функции заменить на вызов специальной first-order функции `apply`, интерпретирующей данный алгебраический тип.

Рассмотрим пример из функции высших порядков `map` и двух колсайтов, создающих лямбда-функции:

[44] Edward Kmett - Type Classes vs. the World.
[45] `https://downloads.haskell.org/ghc/latest/docs/users_guide/exts/rewrite_rules.html`
[46] (youtube) John Hughes - Keynote: How to specify it!
[47] `https://en.wikipedia.org/wiki/Defunctionalization`

```haskell
map :: (Int -> Int) -> [Int] -> [Int]
map f = \case [] -> []; x:xs -> f x : map f xs

example1 xs = map (\x -> x + 1) xs
example2 y xs = map (\x -> x * y) xs
```

Для каждого лямбда-литерала заводим по конструктору, хранящему замыкание. Аппликацию заменяем на вызов `apply`. Таким образом получили функцию `map` первого порядка.

```haskell
data Fun = F1 | F2 Int
apply :: Fun -> Int -> Int
apply df x = case df of F1 -> x + 1; F2 y -> x * y

map :: Fun -> [Int] -> [Int]
map df = \case [] -> []; x:xs -> apply df x : map df xs

example1 xs = map F1 xs
example2 y xs = map (F2 y) xs
```

3.1.8 Эмуляция полиморфизма высших порядков

Далеко не во всех языках есть полиморфизм высшего ранга, но иногда он бывает полезен. Самое распространённое его применение — эмуляция классов типов стрелочных кайндов вроде `Monad`.

Заметим, что типовый конструктор кайнда `Type -> Type` — это функция на типах, принимающая один тип, и возвращающая другой. Применим дефункционализацию, чтобы избежать необходимости параметризовать один типовый конструктор другим Xia, Yallop and White [2014].

Поставим в соответствие типовому конструктору `List` тип-"символ" `ListSym` (для примера используем Kotlin):

```kotlin
class ListSym
```

Заведём тип, соответствующий аппликации символа к типу, хранящий оригинальное значение со стёртым типом:

```kotlin
class Apply<Sym, T>(val value: Any)
```

Установим изоморфизм между изначальным типом, полученным типовой аппликацией конструктора, и новой аппликацией символа:[48]

```kotlin
fun <T> List<T>.to(): Apply<ListSym, T> = Apply(this)
fun <T> Apply<ListSym, T>.from(): List<T> = this.value as List<T>
```

[48] Слева от точки в декларации указывается дополнительный аргумент функции с синтаксисом передачи совпадающим с вызовом метода на объекте. Из тела функции на него можно ссылаться с помощью `this`.

Теперь мы можем объявить интерфейс монад и задать реализацию для списка с помощью объекта-синглтона:

```
interface Monad<M> {
    fun <T> pure(x: T): Apply<M, T>
    infix fun <T, R> Apply<M, T>.bind(k: (T) -> Apply<M, R>): Apply<M, R>
}

object ListMonad : Monad<ListSym> {
    override fun <T> pure(x: T): Apply<ListSym, T> = listOf(x).to()
    override fun <T, R> Apply<ListSym, T>.
            bind(k: (T) -> Apply<ListSym, R>): Apply<ListSym, R> =
        this.from().flatMap { k(it).from() }.to()
}
```

И наконец мы можем писать функции над произвольными монадами:[49]

```
fun <M> Monad<M>.go(x: Apply<M, Int>): Apply<M, Int> =
    x bind { it -> pure(it + 1) } bind { it -> pure(it + 2) }

fun test(xs: List<Int>): List<Int> = ListMonad.go(xs.to()).from()
```

Не лишним будет отметить, что результирующий код выглядит несколько чудовищно. Скорее всего, использование этой техники не окупает себя и нужно выбирать другой стиль программирования.

3.2 Семейства

Идея ad-hoc полиморфизма в том, чтобы в зависимости от типа получать различный код. Семейства начинались как продолжение этой идеи в плоскость данных и типов. Так, ассоциированные синонимы типов — различные типы для различных индексов (типовых параметров) Chakravarty et al. [2005a]. Ассоциированные `data` — различные представления для различных индексов Chakravarty et al. [2005b]. В конце концов эти идеи были обобщены до открытых семейств Schrijvers et al. [2008], потом были введены закрытые Eisenberg et al. [2014].

Можно считать, что семейства[50][51] — это типовые конструкторы, задающие множество типов. Конкретный тип из множества можно выбрать, передав типовой параметр, называемый **индексом**. Сравните с обычными полиморфными конструкторами типа, которые ведут себя одинаково вне зависимости от типовых параметров.

Большое количество интересных примеров использования можно найти, например, в Kiselyov et al. [2010].

[49] Аргументы слева от точки умеют самостоятельно запрыгивать в последующие вызовы. Собственно говоря, они являются неявными параметрами.

[50] `https://ghc.gitlab.haskell.org/ghc/doc/users_guide/exts/type_families.html`

[51] `https://serokell.io/blog/type-families-haskell`

3.2.1 Data families

Data families позволяют выбирать декларацию алгебраического типа в зависимости от типового индекса. Например, для более эффективной реализации структур данных. Это напоминает специализацию шаблонов в C++.

```
data family XList elem
data instance XList () = IntList Int
data instance XList Bool = BoolList ByteArray
```

Единственный способ работать с data family — разместить реализацию в классе типов, чтобы она сопровождала значение с того момента, когда типовой индекс конкретный.

```
class XListOp elem where
  xelem :: elem -> XList elem -> Bool

instance XListOp () where
  xelem () (IntList size) = size > 0

instance XList Bool where
  xelem key (ByteArray bs) = ...

xelemAll :: XListOp elem => XList elem -> [elem] -> Bool
xelemAll xs = all (`xelem` xs)
```

К сожалению, в отличие от специализации шаблонов, нельзя задать определение по-умолчанию вроде `data instance XList a = AList [a]`.

3.2.2 Synonym families

Synonym families (семейства типов) фактически являются функциями на типах. Они бывают открытыми, закрытыми и ассоциированными. Открытые семейства открыты в том смысле, что инстансы можно писать отдельно от декларации, подобно классам типов. Закрытые, наоборот, полностью описываются в одном месте, кейсы упорядочены сверху вниз, что несколько ослабляет ограничения на паттерны.

```
type family Plus (n :: Nat) (m :: Nat) :: Nat where
  Plus Zero m = m
  Plus (Suc n) m = Suc (Plus n m)
```

Чтобы посмотреть, во что вычисляется нечто на уровне типов, можно воспользоваться следующей командой в ghci:

```
ghci> :k! Plus (Suc Zero) (Suc (Suc Zero))
Plus (Suc Zero) (Suc (Suc Zero)) :: Nat
= Suc (Suc (Suc Zero))
```

Ассоциированные семейства работают аналогично, только объявляются в рамках некоторого класса типов, являясь некоторой функциональной альтернативой FunctionalDependencies Jones [2000] (которые выглядят скорее реляционно). Иначе говоря, позволяют поставить в соответствие типу, для которого написан инстанс, другой тип. Например, коллекции — тип её элементов:

```
class Container c where
  type Elem c
  elements :: c -> [Elem c]

instance Container [a] where
  type Elem [a] = a
  elements = id

instance Container ByteString where
  type Elem ByteString = Word8
  elements = ByteString.unpack
```

В современных языках часто встречаются ассоциированные семейства под видом ассоциированных типов[52]. Так, Swift сильно полагается на ассоциированные типы, вовсе не поддерживая дженерики в протоколах (интерфейсах)[53]. В то же время Scala пытается отслеживать, в присутствии экзистенциальных типов (интерфейсов), из какого именно значения пришёл тот или иной ассоциированный тип с помощью path-dependent types Amin et al. [2014].

3.2.3 Инъективные семейства

Семейства типов отличаются от типовых конструкторов примерно так же, как функции от конструкторов данных. Конструкторы пассивны и не редуцируются (`Maybe Int`), в то время как функции вычисляются в какой-то результат (e.g. `F Int` $\sim$ `Bool`). В частности, как и функции, семейства не обязательно инъективны.

Для типовых конструкторов, зная, что сконструированные ими типы эквивалентны, можно вывести, что типовые аргументы эквивалентны тоже. Например:

```
Maybe a ∼ Maybe b ⇒ a ∼ b
```

Очевидно, что для классов типов это свойство по умолчанию не выполняется:

[52] Ассоциированные типы являются фактически экзистенциальными типами, их связывает с ассоциированными семействами логический процесс сколемизации.

[53] (youtube) 2017 LLVM Developers' Meeting: "Implementing Swift Generics"

```haskell
type family NonInjective a where
  NonInjective Int = Double
  NonInjective Char = Double
```

Haskell предоставляет явный синтаксис для объявления инъективных семейств типов, напоминающий функциональные зависимости в классах типов (TypeFamilyDependencies) Stolarek et al. [2015]. Конечно, компилятор проверит, что реализация инъективна. Синтаксис требует связать результат именем через равенство и указать, аналогично FunctionalDependencies, что результат определяет какие-то из типовых индексов семейства:

```haskell
type family InjectiveB a b = r | r -> b
  ...
```

3.2.4 Семейства первого класса

Помимо инъективности, классы типов также не обязательно обладают свойством **generavity**, критически важным для вывода типов, — один и тот же результат не обязательно получен из того же самого семейства:

```haskell
f a ~ g a ⇒ f ~ g
```

Вместе инъективность и generavity — **matchability**. Когда Haskell работает с типом стрелочного кайнда, он подразумевает, что этот тип matchable. Соответственно, семейства не могут передаваться в качестве параметров, а все их вхождения должны быть полностью применёнными ко всем аргументам (fully saturated). Либо нужно явно указать, что семейство возвращает конструктор (в этом конкретном месте семантика зависит от переноса аргументов направо от `::`):

```haskell
type family ToCtor (s :: Symbol) :: Type -> Type where
  ToCtor "maybe" = Maybe
  ToCtor "identity" = Identity
```

Одним из способов обойти это ограничение является дефункционализация семейств Xia, Eisenberg and Stolarek [2014]. Как мы и обсуждали ранее 3.1.7, вместо функции первого класса заводится некоторый символ, обозначающий её, и функция интерпретации, умеющая сделать действие, соответствующее этому символу. В данном случае функцией интерпретации будет открытое семейство `Apply` [Maguire, a, глава 10].

В Haskell ведутся работы[54] по устранению saturation ограничения Kiss et al. [2019]. Для этого нужно различать matchable и unmatchable типовые функции. Это предлагается делать дополнительным индексированием стрелочных кайндов: $\to\equiv\to^{M}$ и $\twoheadrightarrow\equiv\to^{U}$. И, конечно, эти индексы могут быть полиморфными.

[54] GHC proposal: Unsaturated Type Families.

```haskell
data Matchability = Matchable | Unmatchable

hMap
  :: forall (m :: Matchability) (c :: Type -> Constraint)
   . forall (f :: Type ->^m Type) (as :: [Type])
   . All as c => (forall a. c a => a -> f a) -> HList as -> HList (Map f as)
```

3.3 Кайнд `Constraint`

Давно появлялись предложения добавить в GHC поддержку синонимов для констреинтов, семейств констреинтов и т.д Orchard and Schrijvers [2010]. В итоге был предложен[55] и реализован[56] некоторый механизм унификации типов и констреинтов. Таким образом, всё, что работало для типов, стало работать и для констреинтов.

В GHC с ConstraintKinds был добавлен специальный кайнд `Constraint`:

- Класс типов конструирует констреинт: `Monad :: (Type -> Type) -> Constraint`;
- Эквивалентность является констреинтом: `(a ~ b) :: Constraint`;
- Пустой кортеж констреинтов является констреинтом: `() :: Constraint`;
- Кортеж констреинтов является констреинтом: `(Eq a, a ~ b) :: Constraint`.

Теперь, например, мы можем реифицировать словарь как объект языка:[57]

```haskell
data Dict (c :: Constraint) where
  Dict :: c => Dict c
```

Вспомним гетерогенный список, рассмотренный ранее 2.1.5:

```haskell
data HList (tys :: [Type]) where
  HNil :: HList '[]
  HCons :: ty -> HList tys -> HList (ty : tys)
```

Эта структура данных является first-class аналогом variadic generics в C++ или Swift[58] (собственно, смысл вариадиков — не работать `HList`-подобными структурами напрямую). Например, мы можем написать `map` для такой структуры, если все типы удовлетворяют определённому ограничению. Для этого сначала реализуем семейство, генерирующее кортеж констреинтов для каждого типа из списка:

```haskell
type family All (c :: k -> Constraint) (tys :: [k]) :: Constraint where
  All c '[] = ()
  All c (ty : tys) = (c ty, All c tys)

-- All Show [Int, Double] ~ (Show Int, (Show Double, ()))
```

[55] https://gitlab.haskell.org/ghc/ghc/-/wikis/kind-fact
[56] http://blog.omega-prime.co.uk/2011/09/10/constraint-kinds-for-ghc/
[57] https://hackage.haskell.org/package/constraints-0.14.2/docs/src/Data.Constraint.html#Dict
[58] https://github.com/swiftlang/swift-evolution/blob/main/proposals/0398-variadic-types.md

Теперь можем реализовать `map`:

```
hmap :: forall c res tys . All c tys
     => (forall ty . c ty => ty -> res) -> HList tys -> [res]
hmap f = \case
  HNil -> []
  HCons x xs -> f x : hmap @c f xs

ghci> hmap @Show show (HCons (1 :: Int) $ HCons 'a' HNil)
```

Больше такого рода упражнений в гетерогенных конструкциях можно найти в de Vries and Löh [2014].

Констреинты также могут быть параметрически-полиморфными[59] Bottu et al. [2017]:

```
data Rose f x = Rose x (f (Rose f x))

instance (Eq a, forall b. Eq b => Eq (f b)) => Eq (Rose f a) where
  Rose x1 rs1 == Rose x2 rs2 = x1 == x2 && rs1 == rs2
```

3.4 Использование ad-hoc полиморфизма

Часто языки, имеющие что-то напоминающее классы типов, стремятся выразить через них как можно больше других языковых возможностей и полезных техник. Оказывается, это на удивление мощный механизм. Рассмотрим в этом параграфе некоторые избранные примеры.

3.4.1 Сериализация

Классическим примером использования классов типов является сериализация. Проблема в том, что десериализация производится, когда самого объекта ещё нет (ущербный Java подход заполнения объекта дефолтными значениями с последующей мутацией мы не рассматриваем). Поэтому нет возможности написать ООП интерфейс Serializable.

Стандартная библиотека сериализации в Kotlin[60] предоставляет сущность `KSerializer`, которая является интерфейсом для отдельного объекта-сериализатора нашего типа (для эффективности тут используется CPS в виде потоков событий encoder и decoder, вернёмся к этому подходу далее 4.4.1):

```
interface KSerializer<T> {
    fun serialize(encoder: Encoder, value: T)
    fun deserialize(decoder: Decoder): T
}
```

[59] https://downloads.haskell.org/ghc/latest/docs/users_guide/exts/quantified_constraints.html
[60] https://github.com/Kotlin/kotlinx.serialization

Очевидно, чтобы сконструировать сериализатор полиморфного типа, нужны сериализаторы типов-параметров. Что уже отчётливо напоминает классы типов.

```
class PairSerializer(
    keySerializer: KSerializer<K>,
    valueSerializer: KSerializer<V>,
): KSerializer<Pair<K, V>> { ... }
```

Однако в Kotlin нет классов типов, а значит создавать сериализаторы придётся вручную. С этим, однако несколько помогает пачка технологий: `inline fun`, `reified` дженерики, рефлексия и компиляторный плагин библиотеки, который, однако не безопасен с точки зрения типов.

3.4.2 Экзистенциальные типы

Квантор существования в типах является механизмом инкапсуляции [Pierce, 2002, глава 24]. Так, мы можем в типе τ заменить вхождения подтипа σ на переменную α и получить $\exists\alpha.\,\tau'$, где $[\alpha \mapsto \sigma]\,\tau' \equiv \tau$. Таким образом, мы скрыли от пользователя часть типа.

Например, можно с помощью экзистенциальных типов привести разные типы к одной форме и сложить их значения в один список. В Haskell экзистенциальный тип выражают через `data` декларацию, в которой полиморфный конструктор данных и мономорфный тип[61]. Так, тривиальный экзистенциальный тип $\exists\alpha.\,\alpha$ можно получить в Haskell в виде типа `Any`, заданного следующим образом:

```
data Any where
  Any :: forall a . a -> Any  -- логически эквивалентно (exists a . a) -> Any

list :: [Any]
list = [Any 42, Any "Hello", Any (Just Nothing)]
```

В месте деконструирования `Any`, будет доступно значение некоторого неизвестного типа. Очевидно, с таким значением ничего сделать нельзя. Однако, помимо значения, можно положить в конструктор свидетельство о том, что этот неизвестный тип удовлетворяет некоторому классу типов[62].

```
data Has (c :: Type -> Constraint) where
  Has :: c a => a -> Has c
```

Значение типа `Has` свидетельствует о том, что существует некоторый населённый тип `a`, который принадлежит определённому классу типов. Например, рассмотрим `Show`:

[61] Подобно идиоме type erasure в C++.

[62] Подобно Rust fat pointers (пара из указателя на данные и указателя на таблицу виртуальных функций) и ООП объектам.

```haskell
showAll :: [Has Show] -> String
showAll = List.intercalate "," . map \(Has x) -> show x
```

В общем случае, чтобы элиминировать такой тип данных нужны типы высших рангов [Pierce, 2002, глава 24.3]:

```haskell
foldHas :: Has c -> (forall a . c a => a -> b) -> b
foldHas (Has x) k = k x
```

Подробнее можно посмотреть в [Maguire, a, глава 7] и [Pierce, 2002, глава 24].

3.4.3 Разрешение имён

Процесс разрешения имён (name resolution) в языках программирования определяет, с какой программной сущностью связать то или иное употребление имени. Разрешение имён рассматривает импорты, иерархию пространств имён и скоупов, типы выражений. . . Как правило, это сложный процесс, неотделимый от вывода типов.

Однако, в GHC стадия разрешения имён довольно простая и отрабатывает до вывода типов. На её тривиальную суть намекает её название — Renamer — она просто переписывает имена в программе на fully-qualified имена, опираясь на импорты.

С одной стороны, простота — это хорошо. С другой — строгое отделение от вывода типов накладывает неприятное ограничение: типы не могут участвовать в разрешении имён. Наиболее остро эта проблема стоит с метками полей в рекордах. Приходится называть все поля в модуле по-разному, чтобы избежать клешей.

Чтобы заставить разрешение имён зависеть от типов, Haskell снова прибегает к классам типов. А именно, определяется класс типов `IsLabel`, который зависит от символа и ожидаемого типа:

```haskell
class IsLabel (s :: Symbol) a where
  fromLabel :: a
```

Для вызова `fromLabel` есть синтаксический сахар (OverloadedLabels):

```haskell
#name ≡ fromLabel @"name"
```

Теперь разрешение имени `name` будет учитывать тип[63]:

```haskell
data Pet = Pet { name :: String }
instance IsLabel "name" (Pet -> String) where
  fromLabel Pet{ name } = name

data Person = Person { name :: String, pets :: [Pet] }
instance IsLabel "name" (Person -> String) where
  fromLabel Person{ name } = name

ghci> #name pet
```

[63] Чтобы избавиться от ошибки переопределения, нужно включить NoFieldSelectors.

3.4.4 Несинтаксические типовые эквивалентности, System FC

Современный Haskell является синтаксически богатым языком, который, однако, несмотря не многообразие конструкций, транслируется в маленький типизированный внутренний язык. Это язык *System* F_C Sulzmann et al. [2007b], расширяет *System F* (2.1) несинтаксическими эквивалентностями типов. Оказывается, этого достаточно, чтобы поддержать такие возможности Haskell как обобщённые алгебраические типы, ассоциированные семейства типов, функциональные зависимости и т.д.

А именно, вводится встроенный констреинт $\sim$, свидетельствующий о эквивалентности двух типов[64]. Например, тип функции `id` может быть записан таким странным образом:

```
f :: forall a b . a ~ b => a -> b
f = id
```

На самом деле это функция от четырёх параметров: двух типовых параметров, коерции и аргумента. Коерция — это значение размера 0, автоматически выводимое компилятором, которое является свидетельством того, что два соответствующих типа эквивалентны.

Например, GADT из 2.1.4 рассахаривается следующим образом:

```
data Expr ty where
  Const :: Int -> Expr Int
  IsZero :: Expr Int -> Expr Bool
  If :: forall ty . Expr Bool -> Expr ty -> Expr ty -> Expr ty
-- транслируется в
data Expr ty where
  Const :: forall ty . ty ~ Int => Expr ty
  IsZero :: forall ty . ty ~ Bool => Expr Int -> Expr ty
  If :: forall ty . Expr Bool -> Expr ty -> Expr ty -> Expr ty
```

И после паттерн-матчинга по конструкторам, констреинт эквивалентности попадёт в ветку и позволит системе вывода типов сделать необходимые переписывания.

Про вывод типов при наличии локальных предположений можно почитать в классической статье *OutsideIn*(X) Vytiniotis et al. [2011].

Очевидно, что Haskell может населить констреинт эквивалентности следуя рефлексивности, симметричности и транзитивности. Также, компилятор может генерировать новые аксиомы (пользователь напрямую свои аксиомы записать не может). Например, по семейству типов компилятор генерирует аксиомы равенства аппликированного конструктора семейства результирующим типам:

```
type family Plus (n :: Nat) (m :: Nat) :: Nat where
  Plus Zero m = m
  Plus (Suc n) m = Suc (Plus n m)
-- раскроется в
```

[64] https://ghc.gitlab.haskell.org/ghc/doc/users_guide/exts/equality_constraints.html

```
axiom Plus Zero m ~ m
axiom Plus (Suc n) m ~ Suc (Plus n m)
```

3.4.5 Коерции и роли

Haskell имеет поддержку **generative type abstractions** в виде `newtype` деклараций. Эта техника позволяет задавать доменно-специфичные типы, которые во время исполнения не отличимы от оборачиваемых типов, но позволяют различать их. Так, мы можем ввести обёртки для чисел, которые в предметной области представляют собой идентификаторы различных сущностей. Теперь система типов не даст их перепутать.

```
newtype ModuleId = ModuleId Int64
newtype CourceId = CourceId Int64
```

Существует крайне недооценённая практика программирования[65], когда у нас в программе есть чёткая граница, на которой происходит парсинг данных из внешнего мира. После неё сырые неструктурированные данные обогащаются структурой и принимают смысл внутри предметной области. Либо же мы отвергаем эти данные как некорректные. В оставшейся же части программы мы уже пользуемся типизированными данными, свойства которых уже установлены и гарантированы. Например, мы можем быть уверены, что число `ModuleId` строго больше нуля.

Однако, если у нас есть коллекция обёрнутых данных, а мы хотим с ней поработать как с коллекцией сырых, то нам придётся трансформировать коллекцию, несмотря на то, что эта трансформация ничего не делает. Оптимизатор Haskell не справится её элиминировать, потому что работает с типизированным промежуточным представлением и не сможет избавиться от преобразования, меняющего тип.

```
newtype Csv = Csv { unCsv :: String }

concatC :: [Csv] -> Csv
concatC = Csv . concat . map unCsv
```

Поэтому в Haskell есть механизм безопасных коерций между типами, у которых одинаковое представление во время исполнения. Это реализовано с помощью магического класса типов `Coercible`. Его имплементирует компилятор автоматически (см. рис. 12).

```
class Coercible from to where
  coerce :: from -> to
```

Теперь, можем избавиться от лишней трансформации списка:

```
concatC :: [Csv] -> Csv
concatC = coerce concat
```

[65] https://lexi-lambda.github.io/blog/2019/11/05/parse-don-t-validate/

The most important rules that GHC uses to solve Coercible constraints are as follows (the full rules are given in Figure 5):

(1) The *unwrapping rule*:

- For every **newtype** NT = MkNT t, we have Coercible t NT if and only if the constructor MkNT is in scope.

(2) The *lifting rule*:

- For every type constructor TC r p n, where
  - r stands for TC's parameters at a representational role,
  - p for those at a phantom role and
  - n for those at a nominal role,

  if Coercible r1 r2, then Coercible (TC r1 p1 n) (TC r2 p2 n).

(3) Coercible is an equivalence relation:

- The *reflexivity rule*: Coercible a a.
- The *symmetry rule*: If Coercible a b then Coercible b a.
- The *transitivity rule*: If Coercible a b and Coercible b c then Coercible a c.

Рис. 12: Принципы построения инстансов `Coercible` Breitner et al. [2014].

Безопасность коерций обеспечивает **система ролей**. Каждый типовой параметр имеет специальное свойство — роль.

Роль phantom имеют фантомные типовые параметры. Их можно свободно коерсить (нет пререквизитов для инстансов `Coercible`):

```haskell
data Phantom h = Phantom
data NestedPhantom b = MkNP [Phantom b] | SomethingElse

instance Coercible (Phantom a) (Phantom b)
instance Coercible (NestedPhantom a) (NestedPhantom b)
```

Типовой параметр имеет роль representational, если типовой конструктор можно коерсить только при условии, что можно коерсить аргументы:

```haskell
data Maybe a = Nothing | Just a
instance Coerce a b => Coerce (Maybe a) (Maybe b)
```

Роль nominal имеет типовой параметр, если типовой конструктор можно коерсить только при условии, что аргументы эквивалентны. Это требуется, если типовой аргумент индексирует семейство или констреинт.

```haskell
type family F a
data Applied a = Applied (F a)
instance (a ~ b) => Coercible (Applied a) (Applied b)

data ShowDict a where
  ShowDict :: Show a => a -> ShowDict a
instance (a ~ b) => Coercible (ShowDict a) (ShowDict b)
```

Иногда компилятор выводит неправильную роль типовому параметру. Например, если инварианты структуры зависят на конкретную имплементацию какого-то класса типов для типового аргумента, что совершенно не видно в декларации самого типа. В таком случае, роли можно указать явно:

```haskell
type role Map nominal representational
data Map k v = ...
```

Подробнее можно прочитать в Breitner et al. [2014] и [Maguire, a, глава 8].

3.4.6 Type reflection

Рефлексия — это языковой механизм получения информации о типах во время исполнения (на уровне термов). Звучит знакомо, и действительно, Haskell реализует этот механизм через классы типов Peyton Jones et al. [2016].

Библиотека предоставляет магический класс типов `Typeable`, который реализуется компилятором для каждого конкретного типа через `deriving`. Чтобы получить информацию о типе, в скоупе должен быть инстанс `Typeable` для этого типа. Структура типа представлена типом-суммы `TypeRep`, который предоставляет возможность дополнительного типового контроля с помощью обобщённых алгебраических типов данных и типовых тегов.

```haskell
class Typeable a where
  typeRep# :: TypeRep a
```

Например, следующим образом можно получить имя конструктора типа:

```haskell
typeName :: forall a. Typeable a => String
typeName = tyConName $ typeRepTyCon $ typeRep $ Proxy @a

ghci> typeName @Int
```

Упражнение 18 *Объявите класс типов, который позволяет распечатать список типов.*

С помощью структуры представления типа и экзистенциальных типов в Haskell можно эмулировать динамическую типизацию. А именно: любой тип может быть преобразован в `Dynamic`, а потом безопасно преобразован обратно.

```haskell
data Dynamic where
  Dynamic :: Typeable a => a -> Dynamic

fromDynamic :: Typeable a => Dynamic -> Maybe a
```

Это может быть полезно, например, для определения гетерогенного хранилища ключ-значение:

```haskell
data Store = Map Key Dynamic
data Ref ty = Ref Key
get :: Typeable ty => Store -> Ref ty -> Maybe ty
```

3.4.7 Data reflection

Как мы обсуждали ранее, в свойство когерентности гарантирует, что каждому типу в Haskell соответствует ровно один инстанс определённого класса типов. И единственный способ объявить инстанс в Haskell — декларацией на верхнем уровне, то есть он не может зависеть ни от каких локальных данных. Однако в Haskell есть библиотека Data.Reflection[66], которая позволяет создавать локальные инстансы для свежих, чёрной магией сгенерированных[67], типов.

Она пользуется идеей "поднятия значений в типы", обсуждённой нами ранее (см. 3.1.4), но в несколько более общем виде. Вместо заведения классов типов вида `Known_`, вводится один класс типов, индексированный типом термов `terms`, которые спускаются из типов:

```haskell
class Reifies ty terms | ty -> terms where
  reflect :: Proxy ty -> terms
```

Также, с помощью следующей функции, библиотека позволяет сгенерировать свежий тип и инстанс `Reifies`, который по этому свежему типу возвращает данное значение типа `a` (переданное первым аргументом). Поскольку он передаётся в функцию высшего ранга, свежий тип не может утечь из скоупа 2.1.2:

```haskell
reify :: a -> (forall fresh . Reifies fresh a => Proxy fresh -> res) -> res
```

Чтобы воспользоваться нестандартным инстансом класса типов для некого типа `a`, нужно объявить новый тип (например, с помощью `newtype`), содержащий данный, и написать для него нужный инстанс (см, например, Down). Мы не хотим объявлять по новой декларации для каждого случая, поэтому заведём обёртку, похожую на Data.Tagged, которая позволяет добавлять фантомный типовой тег к типу значения. Варьируя тег, можно получить сколь угодно много типов, оборачивающих данный.

```haskell
newtype Wrapped tag a = Wrapped { unwrap :: a }
```

[66] https://www.tweag.io/blog/2017-12-21-reflection-tutorial/
[67] https://www.schoolofhaskell.com/user/thoughtpolice/using-reflection

Объявим тип обёртки `Wrapped tag a` представителем нужного класса типов. Код для реализации будем с помощью `reflect` получать по типу тега в виде честного словаря.

```
data ReifiedOrd a = ReifiedOrd { compare :: a -> a -> Ordering }

instance Reifies tag (ReifiedOrd a) => Ord (Wrapped tag a) where
  compare = coerce $ compare $ reflect $ Proxy @tag
```

Наконец, можем вызвать функцию сортировки, подменив локально порядок на обратный:

```
sort :: Ord a => [a] -> [a]

sortReverse :: forall a . Ord a => [a] -> [a]
sortReverse xs =
  let dict = ReifiedOrd { compare = flip compare } in
  reify dict \(Proxy :: Proxy fresh) ->
    coerce $ sort @(Wrapped fresh a) $ coerce xs
```

3.4.8 Открытые структуры

В динамических языках можно создавать объекты на ходу, последовательно дописывая в них содержимое, и не вводя предварительно декларацию. В Haskell тоже так можно, используя пары для произведений и `Either` для сумм. Например, можно добавить новое поле, создав новую пару: `(oldObj, newField)`.

Однако, такая реализация не оптимальна как с точки зрения эффективности (о более эффективных реализациях можно почитать в [Maguire, a, глава 11]), так и с точки зрения удобства использования. А именно — порядок полей имеет значение и на типах в Haskell нет отношения подтипизации (например, нельзя передать значение с меньшим количеством полей или вариантов). Но можно заметить, что констреинты лишены этих недостатков. Поэтому можно организовывать тип структуры данных, например, таким образом:

```
(Int, Double) заменяем на (Member Int d, Member Double d) => Prod d
```

3.4.9 Исключения и открытая иерархия

Важный аспект работы с ошибками заключается в том, что многие из них обрабатываются единообразно. Таким образом, ошибки должны образовывать иерархию наподобие той, которая в ООП языках получается с помощью наследования, чтобы иметь возможность реагировать сразу на группу ошибок одним кодом. Так, возникает задача моделирования подобной иерархии в Haskell.

Более того, статически типизированные ошибки это активная область исследований, будем говорить об этом в рамках систем эффектов (см. далее **??**). Классические исключения же

динамически типизированные. Особенно хорошо этот вариант подходит для ошибок программиста, которые по-хорошему не должны обрабатываться в программе кроме как закрытием ресурсов.

Поддержка исключений присутствует в системе исполнения Haskell как простого и привычного способа обработки исключительных ситуаций: ошибок программиста, исполнения неполного паттерн-матчинга, асинхронных системных сигналов Marlow et al. [2001]. . . Исключения динамически типизированные и образуют иерархию Marlow [2006]. Если породить исключение может и чистый код, так как $\bot$, по семантике Haskell, населяет любой тип, то поймать исключение можно только[68] в `IO` Jones [2001], используя специальные примитивы языка.

Чтобы сделать тип исключением, нужно объявить инстанс `Exception` для него:

```
class (Typeable a, Show a) => Exception a where
  toException :: a -> SomeException
  toException = SomeException

  fromException :: SomeException -> Maybe a
  fromException (SomeExcetion e) = cast e
```

Где `SomeException` — это экзистенциальная обёртка наподобие `Dynamic` (см. 3.4.6), в которую заворачивается конкретный тип исключения. Ловя `SomeException`, можно поймать любое исключение (`cast` всегда сработает).

```
data SomeException where
  SomeException :: Exception a => a -> SomeException

instance Exception SomeException
```

Система исполнения Haskell предоставляет интринсики для кидания и ловли исключений, обернём их для поддержки любого `Exception` типа:

```
throw :: Exception e => e -> a
throw = primThrow (toException e)

catch :: Exception e => IO a -> (e -> IO a) -> IO a
catch io handler = io `primCatch` \e -> case fromException e of
  Nothing -> throw e
  Just e' -> handler e'

ghci> throw "error" `catch` (e :: String) -> putStrLn e
```

В простейшем случае свой тип исключения можно реализовать в две строчки. Чтобы его поймать, нужно либо ловить сам этот тип, либо `SomeException`, потому что для них обоих `fromException` на объекте вида `SomeException MyError` вернёт `Just`[69].

[68] (stackoverflow) Why can Haskell exceptions only be caught inside the IO monad?

[69] `instance Exception SomeException where fromException = Just`

```haskell
data MyError = MyError deriving (Show, Typeable)
instance Exception MyError
```

Добавим исключение ArithException и ещё один более общий тип исключений между ним и SomeException — SomeArithException (таким образом, будет три способа поймать ArithException). Для этого сделаем SomeArithException экзистенциальной обёрткой, а каждое исключение типа ArithException будем автоматически оборачивать в неё. В fromException на каждом уровне вложенности будем пытаться получить оборачивающий конструктор рекурсивным вызовом.

```haskell
data SomeArithException where
  SomeArithException :: Exception a => a -> SomeArithException

-- SomeException - базовый (реализация по умолчанию)
instance Exception ArithException

data DivisionByZero = DivisionByZero deriving Show
instance Exception DivisionByZero where
  toException = toException . SomeArithException
  fromException e = do
    SomeArithException e' <- fromException e
    cast e'
```

Так, во время бросания DivisionByZero будет конструироваться объект вида:

```haskell
SomeException (SomeArithException DivisionByZero)
```

Реализация fromException для конкретного типа умеет убедиться в наличии соответствующего конструктора в результирующем объекте исключения.

3.4.10 Легковесные частичные стек-трейсы

Забавной эксплуатацией классов типов в Haskell являются легковесные частичные стек-трейсы[70]. Вообще для сбора трейсов нужна поддержка рантайма, что в случае Haskell усложняется ещё и тем, что модель вычислений, редукция графов, реальных трейсов не содержит и их приходится эмулировать. Мы же получим трейсы без поддержки рантайма.

В стандартной библиотеке определён констреинт GHC.Stack.HasCallStack, позволяющий получить информацию о месте вызова функции. Эту информацию фактически размещает компилятор в процессе вывода инстансов. Если в месте вызова доступна информация с уровня выше, компилятор распространяет её дальше. Таким образом, доступна информация только на определённую глубину стека вызовов.

[70] https://downloads.haskell.org/ghc/latest/docs/users_guide/exts/callstack.html

```haskell
myHead :: HasCallStack => [a] -> a
myHead []     = error "empty"
myHead (x:xs) = x

bad :: Int
bad = myHead []
```

```
ghci> bad
*** Exception: empty
CallStack (from HasCallStack):
  error, called at Bad.hs:8:15 in main:Bad
  myHead, called at Bad.hs:12:7 in main:Bad
  -- no information about bad call site here
```

HasCallStack — это просто имплисит (см. 3.1.2), про который знает компилятор:

```haskell
type HasCallStack = (?callStack :: CallStack)
```

3.4.11 Кастомизируемые ошибки типизации

При программировании сложных с точки зрения типов библиотек, желательно предоставлять пользователям более информативные ошибки типизации, чем ошибки по умолчанию. Для этого в GHC есть механизм в GHC.TypeLits, позволяющий сконструировать специальный тип, информация из которого попадёт в сообщение об ошибке. Например, этот тип можно вернуть из synonym family при некорректном наборе аргументов. Или же можно воспользоваться constraint trick (см. 3.1.3) и разместить такой тип в качестве посылки в инстансе. Если инстанс подошел и компилятор начал обрабатывать ограничения слева, значит, что-то пошло не так [Maguire, a, глава 12].

```haskell
instance (TypeError
  ( Text "Attempting to show a function of type "
    :<>: Text "'" :<>: ShowType (a -> b) :<>: Text "'"
    :$$: Text "Did you forget to apply an argument?"
  )) => Show (a -> b) where
  show = undefined -- реализация не важна, до исполнения дело не дойдёт
```

4 Типы данных

В этой главе собраны некоторые общие знания о типах. Также, мы получим различные эквивалентные представления рекурсивных типов данных (иначе говоря, коллекций). Многие концепции являются частными случаями этого многообразия.

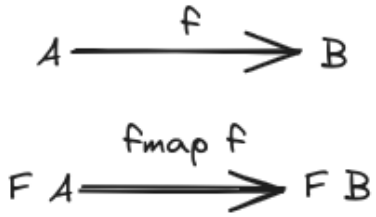
Разделы 4.1, 4.2 в основном следуют [Maguire, a, глава 1].

4.1 Вариантность

В этом параграфе мы будем рассматривать тему с точки зрения программирования [Maguire, a, глава 3], не отдавая должного теории категорий. Восполнить пробел можно с помощью замечательной статьи, написанной в жанре пьесы Hinze et al. [2012].

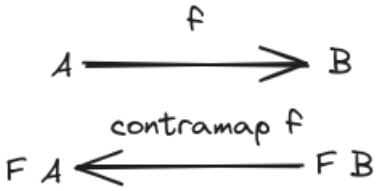
Ковариантный функтор — пара из некоторого типового конструктора `F` и операции на функциях `fmap :: (a -> b) -> (F a -> F b)`. Плюс законы о том, что `fmap` уважает `id` и композицию.

```haskell
class Functor f where
  fmap :: (a -> b) -> (f a -> f b)
```



Контравариантный функтор — пара из типового конструктора и операции на функциях, разворачивающей стрелку. Плюс соответствующие законы.

```haskell
class Contravariant f where
  contramap :: (a -> b) -> (f b -> f a)
```



Типовой конструктор можно объявить ковариантным или контравариантным функтором (или никаким из них) относительно некоторого типового параметра в зависимости от вида

декларации соответствующих конструкторов данных. А именно, от знака позиций, в которых входит этот типовой параметр в тип.

Разовьём интуитивное понимание знаков позиций. Тип `A` входит в положительной позиции в `B` если его значение можно извлечь из `B`. И наоборот, тип `A` входит в отрицательной позиции, если его значение нужно, наоборот, предоставить. Рассмотрим знаки позиций типов в базовых типовых конструкторах:

Тип	знак позиции `A`	знак позиции `B`
`Either A B`	$+$	$+$
`(A, B)`	$+$	$+$
`A -> B`	$-$	$+$

Действительно, из суммы и произведения можно извлечь компоненты с помощью паттерн-матчинга, а из стрелки можно получить правый тип аппликируя её к аргументу. В то же время значение типа слева от стрелки нужно предоставить.

На плюс и минус действуют интуитивные алгебраические законы при рассмотрении более сложных типов. Рассмотрим на примере `f :: ((A, B) -> C) -> (D, E)`.

- Плюс на плюс даёт плюс. Действительно, нужно лишь применить две элиминации вместо одной, чтобы получить заветный тип. В нашем примере, чтобы получить `D`, нужно сначала аппликировать функцию, а потом разобрать пару.
- Плюс на минус (и наоборот) даёт минус. Действительно, `C` нам нужно предоставить: `f (\ab -> provideC)`.
- Минус на минус даёт плюс. Пару `(A, B)` нам предоставляют: `f (\ab -> ...)`.

Упражнение 19 *Убедитесь что плюс на минус даёт минус.*

Возвращаясь к функторам, если типовой параметр входит в декларацию только в положительных позициях, типовой конструктор можно объявить ковариантным функтором относительно этого параметра. Если в только в отрицательных — контравариантным функтором. Если в обоих, то никаким функтором объявить нельзя. Соответственно, будем называть типовые параметры ковариантными, контравариантными и инвариантными.

Упражнение 20 *Объявите* `instance Contravariant F` *для* `data F a = L (a -> ()) | R Int`.

Таким образом, можно понимать ковариантный функтор как вычисление, результат которого можно пост-обработать, а контравариантный функтор — как вычисление, аргументы которого можно пред-обработать.

Тип от двух положительных параметров можно объявить **бифунктором**:

```
class Bifunctor f where
  bimap :: (a -> c) -> (b -> d) -> f a b -> f c d
```

Тип от двух параметров, положительного и отрицательного, — **профунктором**:

```haskell
class Profunctor p where
  dimap :: (c -> a) -> (b -> d) -> p a b -> p c d
```

Профункторы являются некоторыми обобщениями функциональной стрелки. Например, если у нас есть SQL запрос, который по данным возвращает результат, его можно объявить профунктором с семантикой — добавить пред-обработку входных данных и пост-обработку выходных:

```haskell
dimap serialize deserialize (query :: Sql Text Text) :: Sql Age [User]
```

Также понятие вариантности часто встречается в объектно ориентированных языках (да и вообще в теории подтипизации) для обозначения возможности дополнить отношение подтипизации на полиморфные типы.

Действительно, **отношение подтипизации** `B <: A` говорит о том, что значение типа `B` безопасно использовать в позиции, где ожидается значение типа `A`. Иначе говоря, существует функция `upcast :: B -> A`. Если типовой конструктор `F a` ковариантен относительно параметра `a`, то по `upcast` найдётся `upcast' :: F B -> F A`. То есть отношение подтипизации также автоматически включает `F B <: F A`. Контравариантный случай аналогично.

Упражнение 21 *Убедитесь в вашем любимом языке с подтипизацией и поддержкой вариантности, что минус на минус даёт плюс.*

4.2 Изоморфизм

Пусть нам нужно спроектировать функцию или модель данных. Мы начинаем с декларации типа, как её выбрать и из каких вариантов? Для начала поймём, когда два типа взаимозаменимы, для этого рассмотрим понятия изоморфизма.

Два типа `A` и `B` называются **изоморфными** (обозначают $A \cong B$) тогда и только тогда, когда существует такая пара функций `to :: A -> B` и `from :: B -> A`, что[71]

```haskell
to . from = id
from . to = id
```

Иначе говоря, между обитателями таких типов можно установить взаимно-однозначное соответствие. Легко понять, что со смысловой точки зрения не принципиально, какой из изоморфных типов использовать — их можно заменять друг на друга, добавляя вызовы функций перехода. Такие два типа заключают в себе одинаковое "количество информации". Например, типы `Bool` и `Maybe ()` в этом смысле совершенно взаимозаменимы. Покажем это, предъявив пару взаимообратных функций[72]:

[71] Под равенством термов можно понимать разное, например, $\alpha\beta\gamma$-эквивалентность. Мы будем пользоваться **экстенсиональным равенством** для функций — две функции равны, когда равны их результаты на всех входах. https://ncatlab.org/nlab/show/function+extensionality

[72] Нужно не забыть показать взаимообратность функций, но это делается тривиально перебором входов (может быть с помощью индукции) и редукцией.

```haskell
to :: Bool -> Maybe ()
to b = if b then Just () else Nothing

from :: Maybe () -> Bool
from m = case m of Nothing -> False; Just () -> True
```

Несмотря на смысловую взаимозаменимость, для кодирования информации о том, передал ли пользователь программе определённый флаг, мы, скорее всего, воспользуемся типом `Bool` ввиду нефункциональных соображений о читабельности кода. Аналогично можно рассматривать соображения эффективности.

С категорным взглядом на происходящее можно ознакомиться в Hinze and James [2010]. Мы же придерживаемся теоретико-множественной интерпретации типов.

4.2.1 Кардинальность: суммы, произведения, экспоненты

Типы можно воспринимать как синтаксис для записи множеств, а населяющие их термы — как синтаксические записи элементов этих множеств. Так терм `(True, False)` — запись элемента множества пар, записываемого в синтаксисе типов как `(Bool, Bool)` (вместо математического $\mathbb{B} \times \mathbb{B}$). Или же терм `\x -> x + 1` является записью функции прибавляющей единицу из множества функций над целыми числами, записываемого как `Integer -> Integer` (вместо математического $\mathbb{Z} \to \mathbb{Z}$).

Заметим, что два типа изоморфны, если соответствующие им множества имеют одинаковое количество элементов. Более того, таких изоморфизмов $n!$ в случае конечности множеств. Научимся определять количество таких элементов. С помощью $|\cdot|$ будем записывать **кардинальность** типа — количество элементов в соответствующем множестве.

Тип и его декларация	кардинальность
`data Void`	0
`data Unit = Unit`[73]	1
`data Bool = False | True`	2

Идея алгебраических типов данных в том, что сложные типы можно строить из простых с помощью операции $+$ (“или”) и операции $\times$ (“и”)[74]:

Тип	кардинальность
`data Either a b = Left a | Right b`	$\lvert a\rvert + \lvert b\rvert$
`data Pair a b = Pair a b`	$\lvert a\rvert \times \lvert b\rvert$

Посчитаем количество обитателей различных типов (вы можете убедиться в справедливости заключения перебрав все термы вручную):

- `|Either Unit (Eigher Bool Bool)| = |Unit| + (|Bool| + |Bool|) = 5`.

[73] `Unit` записывается в Haskell с помощью специального синтаксиса `()`, означающем как бы пустой кортеж.
[74] https://stanford-cs242.github.io/f18/lectures/02-2-algebraic-data-types.html

- `Pair (Either Bool Unit) (Pair Unit Void)| = 0` — тип `Void` не населён, как и кортеж, его включающий.
- Если `data Example = FirstAlternative Bool | AnotherOne Unit Bool Bool`, то `|Example| = |Bool| + |Unit| * |Bool| * |Bool| = 2 + 1 * 2 * 2 = 6`.

Функциональную стрелку называют экспоненциальным типом. Действительно, комбинаторно количество обитателей `A -> B` вычисляется как

$$|A \to B| = |B|^{|A|}$$

Так как же проектировать типы? Тому есть несколько соображений:

- В типе должно быть не меньше элементов, чем в предметной области, все необходимые объекты были представимы.
- В типе должно быть как можно меньше элементов, которых нет в предметной области, чтобы пространство ошибок было минимальным.
- Далее среди изоморфных типов выбирается оптимальный исходя из нефункциональных требований.

Прежде чем работать с некоторым объектом предметной области, информацию о нём, в соответствии со вторым правилом, следует привести в максимально структурное представление, дающее наибольшее количество гарантий[75].

4.2.2 Алгебраическое представление типа

Как мы увидели выше, чтобы показать наличие изоморфизма между двумя типами можно либо предъявить пару взаимообратных функций, либо показать, что кардинальности этих двух типов совпадают. В этом разделе мы научимся сопоставлять типу некоторую алгебраическую запись, отражающую его структуру и кардинальность. Так, мы сможем синтаксическими преобразованиями формул получать эквивалентные записи, из которых будем восстанавливать типы, заведомо изоморфные данному[76].

В основу алгебраического представления положим вычисление кардинальности типов. Фактически мы забываем несущественную для изоморфизма информацию об именах конструкторов данных и конструкторов типов, то есть переходим к структурной типизации.

Тип	алгебраическая формула
`data Void`	0
`data Unit = Unit`	1
`data Bool = False | True`	$1 + 1$ (обозначим как 2)
`data Maybe a = Nothing | Just a`	$1 + a$
`data Either a b = Left a | Right b`	$a + b$
`data Pair a b = Pair a b`	$a \times b$
`a -> b`	b^a

[75] https://lexi-lambda.github.io/blog/2019/11/05/parse-don-t-validate/

[76] https://codewords.recurse.com/issues/three/algebra-and-calculus-of-algebraic-data-types

sums

$$X + 0 \cong X \quad X + Y \cong Y + X$$
$$(X + Y) + Z \cong X + (Y + Z)$$

products

$$X \times 1 \cong X \quad 0 \times X \cong 0 \quad X \times Y \cong Y \times X$$
$$(X \times Y) \times Z \cong X \times (Y \times Z)$$

product over sum

$$(X + Y) \times Z \cong (X \times Z) + (Y \times Z)$$

exponentials

$$1^X \cong 1 \quad X^0 \cong 1 \quad X^1 \cong X$$
$$Z^{X+Y} \cong Z^X \times Z^Y \quad (Z^Y)^X \cong Z^{X \times Y}$$

exponential over product

$$(Y \times Z)^X \cong Y^X \times Z^X$$

Рис. 13: Законы школьной алгебры ностальгии ради Hinze and James [2010].

Упражнение 22 *Запишите в алгебраическом виде следующий тип:*

```
data T a b = Undefined | Defined a (a -> b)
```

В качестве отношения эквивалентности, будем использовать изоморфизм соответствующих типов. В такой интерпретации, классические свойства алгебраических операций сохраняются (рис. 13). Действительно, например:

```
-- (c^b)^a ≅ c^(a×b)
to :: (a -> b -> c) -> (a, b) -> c
to = uncurry
from :: ((a, b) -> c) -> a -> b -> c
from = curry
```

Упражнение 23 *Покажите, что* $(a + b) + c \cong a + (b + c)$.

Упражнение 24 *Покажите, что* $c^{a+b} \cong c^a \times c^b$.

Интересным наблюдением может быть то, что функции можно использовать как структуры данных, в соответствие с изоморфизмом $c^{a+b} \cong c^a \times c^b$. Действительно, в таком случае аргумент функции выступает индексом (его кардинальность должна совпадать с размером коллекции).

```
-- a × a ≅ a^2
get :: (a, a) -> (Bool -> a)
get (x, y) idx = if idx then x else y
tabulate :: (Bool -> a) -> (a, a)
tabulate f = (f True, f False)
```

Каноническим предствлением типа (canonical representaion) называют сумму произведений типов:

$$\sum_i \prod_j t_{ij}$$

Каноническое представление является своего рода нормальной формой, в которой можно записывать алгебраические типы (любой алгебраический тип можно по правилам привести к ней). Легко узнать в нём вид `data` деклараций в Haskell.

Удивительно, но даже производная алгебраического типа имеет вполне понятную семантику. Это контекст зиппера (zipper), структуры данных наподобие итератора, позволяющей навигироваться по структуре данных произвольной формы Huet [1997], McBride [2001], Abbott et al. [2003].

4.3 Рекурсивные типы

Рекурсивные типы, наподобие рекурсивным термам, могут включать себя в своих определениях. Иначе говоря, рекурсивный тип изоморфен какому-то другому типу, в который он сам входит как подтип.

```
1 fac n = if n <= 1 then 1 else n * fac (n - 1)
2 data Nat = Zero | Suc Nat
```

Мы в основном посмотрим на рекурсивные типы с практической точки зрения. Однако, их формальное теоретико-типовое описание, теоретико-категорная и теоретико-множественная интерпретации представляют отдельный интерес [Pierce, 2002, часть 4].

4.3.1 Просто список

Рассмотрим классический функциональный список. Список это либо коллекция из нуля элементов, либо одного, либо двух... Алгебраически это запишется следующим образом:

$$L = 1 + a + a^2 + a^3 + \dots$$

Фактически получили тип с бесконечной записью. Поработаем с ним как с формальным рядом. Вынесем a за скобки:

$$L = 1 + a \times (1 + a + a^2 + \dots)$$

Заметим, что выражение в скобках представляет собой список, получим такое рекурсивное уравнение[77]:

$$L = 1 + a \times L$$

Легко видеть, что это на самом деле знакомое нам определение списка из Haskell:

[77] Либо можно получить то же самое, заметив, что мы имеем дело с рядом Тейлора `https://codewords.recurse.com/issues/three/algebra-and-calculus-of-algebraic-data-types`.

```haskell
data List a = Nil | Cons a (List a)
```

Получим конечное нерекурсивное представление типа L. Фактически, нам нужно получить тип изоморфный типу, включающему в себя исходный:

$$L \cong 1 + a \times L$$

Расширим язык типов абстракцией (полиморфизмом) и решим полученное рекурсивное уравнение в стиле λ-исчисления, с помощью некоторого комбинатора неподвижной точки:

$$L = FIX\ \lambda r.1 + a \times r$$

Закодируем это на Haskell. В качестве комбинатора рекурсии возьмём `data FixList a`:

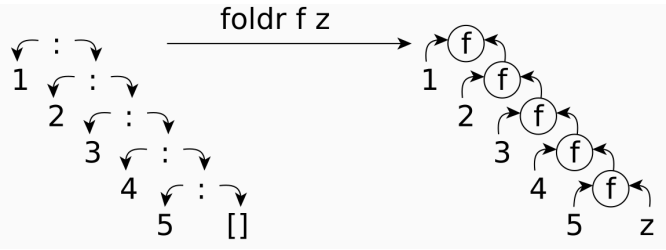
```haskell
type Shape a r = Either () (a, r) -- λar.1 + a × r
data FixList a = In (Shape a (FixList a))
-- FixList a ≅ Shape a (FixList a) ≅ Shape a (Shape a (FixList a)) ≅ ...

example :: FixList Int -- [1, 2, 3]
example = In (Right (1, In (Right (2, In (Right 3, Left ())))))
```

Таким образом, мы разделили определение списка на две части: одна отвечает за форму типа, другая — за рекурсию[78]. Форму можно переиспользовать, в определении свёртки:

```haskell
foldr :: (Either () (a, r) -> r) -> FixList a -> r
foldr phi (FixList shape) = case shape of
  Left () -> phi (Left ())
  Right (x, xs) -> phi (Right (x, foldr phi xs))
-- сравните с классическим определением
foldr :: r -> (a -> r -> r) -> [a] -> r
foldr ini f list = case list of
  [] -> ini
  x:xs -> f x (foldr ini f xs)
```

Действительно, свёртка в общем смысле позволяет заменить каждый конструктор данных в дереве на некоторую функцию. В результате получается вычисление, имеющее доступ ко всему содержимому структуры данных, и возвращающее результат агрегации:



[78] Техника абстрагирования по "рекурсивной ссылке на себя" называется **открытой рекурсией**. Так, мы предоставляем пользователю больше контроля. Например, на уровне термов комбинатор рекурсии может подсчитывать количество рекурсивных вызовов и оборвать вычисление при привышении какого-то лимита.

То есть количество и типы аргументов свёртки зависят от “формы” сворачиваемого типа. В случае списка требуется нольарная функция вместо `Nil` и бинарная вместо `Cons`. Мы же эти две функции представляем в как одну с помощью типа-формы. Сравните:

```
foldr                   0 (      \e  r  -> e + r)
foldr (\case Left () -> 0; Right (e, r) -> e + r)
```

4.3.2 Неподвижная точка функтора

Абстрагируем `FixList` по типу-форме:

```
newtype Fix :: (Type -> Type) -> Type
newtype Fix f = In { out :: f (Fix f) }

data ListF a r = Nil | Cons a r
type List a = Fix (ListF a)
```

Упражнение 25 *Какие типы будут у* `In` *и* `out`*?*

Можно показать, что `[a]` $\cong$ `List a`:

```
to :: [a] -> List a
to = \case
  [] -> In Nil
  x:xs -> In $ Cons x (to xs)

from :: List a -> [a]
from (In shape) = case shape of
  Nil -> []
  Cons x xs -> x : from xs
```

Тип формы можно сделать функтором по последнему параметру. Это позволит нам в дальнейшем заменять вхождения поддеревьев на что-то полезное.

```
instance Functor (ListF a) where
  fmap :: (rec -> other) -> ListF a rec -> ListF a other
  fmap f = \case
    Nil -> Nil
    Cons x xs -> Cons x (f xs)
```

Упражнение 26 *Выразите следующее дерево как неподвижную точку функтора. Объявите инстанс функтора для типа-формы.*

```
data Tree a = Leaf a | Node a (Tree a) (Tree a)
```

4.3.3 Схемы рекурсии

Подобно тому, как структурное императивное программирование, в сравнении с беспорядочным использованием `goto`, помогает рассуждать о программах, так **схемы рекурсии** позволяют алгебраически описывать свойства рекурсивных функций, в отличие от "неструктурной" рекурсии[79] Meijer et al. [1991], Meijer and Hutton [1995].

Глобальная идея состояла в том, чтобы формулировать требуемые свойства и вычислять нужные программы подобно тому, как математики находят решения дифференциальных уравнений. Однако, эта идея не нашла нужного развития и применения (однако, владеть алгебраическим подходом в целом полезно Maguire [b]). Тем не менее это знание, с одной стороны, даёт более глубокое понимание рекурсии, с другой, пригодится нам для рассмотрения и решения главной проблемы этого курса — expression problem.

Универсальная свёртка называется **катаморфизмом**. Катаморфизм сначала рекурсивно сворачивает поддеревья, добираясь к ним с помощью `fmap`, и оставляет результаты свёртки типа `a` вместо бывших вхождений поддеревьев. Получается значение типа `f a`, где `f` — какой-то функтор формы. Далее применяется функция типа `f a -> a`, которая определяет, как свернуть один слой рекурсивной структуры, когда поддеревья уже свёрнуты.

```
cata :: Functor f => (f a -> a) -> Fix f -> a
cata phi = phi . fmap (cata phi) . out
```

Иначе говоря, `cata` заменяет все конструкторы `In` на функцию `phi`, которая содержит информацию о том, как реагировать на разные варианты конструкторов в дереве:

```
example           = In  (Cons 1 (In  (Cons 2 (In  (Cons 3  (In  Nil))))))
cata phi example =β phi (Cons 1 (phi (Cons 2 (phi (Cons 3 (phi Nil))))))
```

Например, сумма списка будет выглядеть следующим образом (рис. 14):

```
sum :: List Int -> Int
sum = cata \case
  Nil -> 0
  Cons x result -> x + result
```

Функцию `f a -> a` называют **f-алгеброй**. Действительно, если в качестве функтора `f` взять сигнатуру алгебры, а в качестве `a` носитель, то f-алгебра будет задавать некоторую интерпретацию сигнатуры:

```
data MonoidSig carrier = Mempty | Mappend carrier carrier

interpretSig :: MonoidSig Int -> Int
interpretSig = \case Mempty -> 0; Mappend l r -> l + r
```

Упражнение 27 *С помощью какой алгебры можно скопировать структуру данных?*

[79] https://reasonablypolymorphic.com/blog/recursion-schemes/index.html

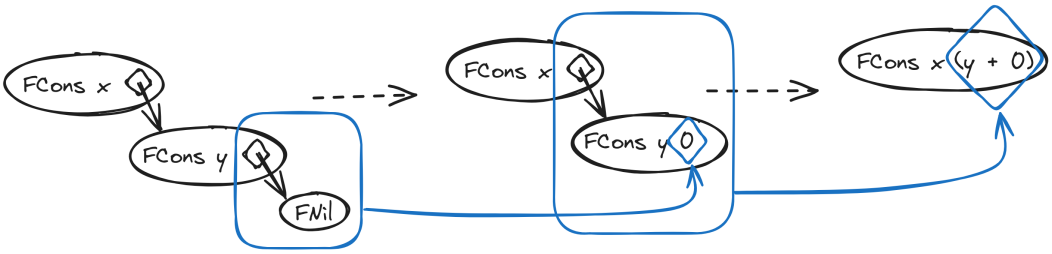


Рис. 14: Катаморфизм переиспользует вхождения рекурсивных поддеревьев, чтобы хранить результаты свёртки соответствующих поддеревьев.

Упражнение 28 *С помощью какой алгебры можно распечатать список в строчку?*

В противоположность универсальной свёртке, можно построить **анаморфизм** — универсальную развёртку (аналогично `unfold` для списка). Здесь f-коалгебра (стрелочка в обратную сторону) показывает, как из некоторого значения-зерна получить один слой структуры данных, где вместо рекурсивных ссылок будут зёрнышки, из которых потом прорастут поддеревья. Анаморфизм как раз сначала разворачивает один слой, а потом рекурсивно разворачивает все поддеревья:

```
ana :: Functor f => (s -> f s) -> s -> Fix f
ana psi = In . fmap (ana psi) . psi
-- сравните с классическим определением развёртки списка
unfoldr :: (s -> Maybe (a, s)) -> s -> [a]
```

Упражнение 29 *Реализуйте анаморфизм, строящий список от* 0 *до заданного* n*.*

Также вводят **гиломорфизм (hylomorphism)**, которые позволяют описать произвольное рекурсивное вычисление. Гиломорфизм задаётся как композиция анаморфизма и катаморфизма. Сначала анаморфизм строит явное дерево, представляющее собой дерево вызовов некоторой рекурсивной процедуры, затем катаморфизм сворачивает его в результат.

```
hylo :: Functor f => (a -> f a) -> (f b -> b) -> a -> b
hylo psi phi = cata phi . ana psi
```

Например, вычисление факториала может быть реализовано следующим образом:

```
fac n = hylo
  (\n -> if n > 0 then Cons n (n - 1) else Nil)
  (\case Nil -> 1; Cons n acc -> n * acc)
```

Можно ввести ещё много различных рекурсивных схем[80] и описать их свойства. Однако, мы пока остановимся.

Интересный факт: катаморфизм представляет собой теоретико-множественный принцип индукции, а анаморфизм — коиндукции, если в качестве вложения множеств взять функцию [Pierce, 2002, глава 21].

[80] `https://wiki.haskell.org/Zygohistomorphic_prepromorphisms`

4.4 Всё через свёртки

Оказывается, что с помощью катаморфизма можно получить изоморфизм между структурами данных и их свёртками: `Fix f ≅ forall a . (f a -> a) -> a`.

```
to :: Functor f => Fix f -> (forall a . (f a -> a) -> a)
to = flip cata

from :: (forall a . (f a -> a) -> a) -> Fix f
from g = g In
```

Например, следующие два списка эквивалентны (все конструкторы `In` заменяем на данную алгебру):

```
data ListF elem rec = Nil | Cons elem rec

xs1 :: Fix (ListF Int)
xs1 = In (Cons 1 (In (Cons 2 (In (Cons 3 (In Nil))))))

xs2 :: (ListF Int a -> a) -> a
xs2 = \alg -> alg (Cons 1 (alg (Cons 2 (alg (Cons 3 (alg Nil))))))

ghci> xs2 @Int \case Nil -> 0; Cons x acc -> x + acc
6
```

Теперь избавимся от функтора формы. Это нерекурсивный тип, который можно представить в канонической форме (см. 4.2.2):

$$f\ a \cong \sum_i \prod_j (t_{ij}\ a)$$

Тогда алгебра может быть записана следующим образом:

$$f\ a \to a \cong a^{\sum_i \prod_j (t_{ij}\ a)} \cong \prod_i a^{\prod_j (t_{ij}\ a)}$$

Остались произведения, от которых можно избавиться с помощью каррирования, и получить **Church encoded** структуры данных. Например, для списка имеем:

$$\texttt{(ListF elem a -> a) -> a} \cong a^{a^1 + elem \times a} \cong a^a \times a^{elem \times a} \cong (a^a)^{\left((a^a)^{elem}\right)}$$
$$\cong \texttt{a -> (elem -> a -> a) -> a}$$

Мы получили не что иное, как список Чёрча[81][82]. Структуру данных, без единого конструктора![83] Перепишем знакомый нам список ещё раз:

[81] https://en.wikipedia.org/wiki/Church_encoding

[82] https://okmij.org/ftp/tagless-final/course/Boehm-Berarducci.html

[83] На самом деле в этом нет ничего удивительного, если вспомнить, что функции первого класса представляются как замыкания, содержащие данные. Мы получили тот же односвязный список, только на замыканиях.

```haskell
xs3 :: a -> (Int -> a -> a) -> a
xs3 = \ini f -> f 1 (f 2 (f 3 ini))
```

Упражнение 30 *Какая знакомая вам стандартная функция работы со списками по* `data` *списку возвращает список Чёрча?*

Попробуем интуитивно понять, что это всё значит. Заметим, что список Чёрча принимает функции, соответствующие веткам паттерн-матчинга или аргументам сворачивающей функции. Вместо конструкторов же сразу вызываются соответствующие функции. То есть, вместо того, чтобы создать структуру данных и доставить её к месту деконструирования (паттерн-матчингу), мы *доставляем место деконструирования к месту конструирования* и оказывается, что ничего конструировать по итогу и не надо.

Мы уже работали с обратным процессом, дефункционализацией, когда функции первого класса превращались в вызовы конструкторов, а их тела — в ветки паттерн-матчинга 3.1.7. Можно заметить, что тут мы имеем дело с обратным процессом, **рефункционализацией**, когда вместо вызовов конструкторов сразу вызывается соответствующий интерпретирующий код Danvy and Millikin [2009].

С технической точки зрения, мы поняли, что условное ветвление (if, паттерн-матчинг, конструкция switch) и виртуальные вызовы (вызовы замыканий, методов на интерфейсах) взаимозаменимы.

Стоит также отметить, что изоморфизм `Fix f` $\simeq$ `forall d . (f d -> d) -> d` является обобщением изоморфизма `a` $\sim$ `forall r . (a -> r) -> r` (при `f = Const a`), который является основой CPS трансформации, рассматриваемой нами далее в главе 6.

4.4.1 Deforestation & list fusion

В функциональном программировании мы строим новые функции путём композирования имеющихся, более простых, функций. Этот подход позволяет переиспользовать реализованную функциональность, снижая сложность кода и вероятность ошибок. Однако, он может приводить к излишним накладным расходам на аллокацию и деконструирование промежуточных структур данных.

Например, сравните следующие две реализации. Первая предпочтительна с точки зрения качества кода, однако она порождает промежуточный список во время работы:

```haskell
all p xs = and (map p xs)
-- или fused версия
all p [] = True
all p (x:xs) = p x && all p xs
```

Очевидно, это задача оптимизирующего компилятора превращать хороший код с абстракциями в быстрый код. Оптимизация, избавляющая программы от (промежуточных) структур данных (деревьев) называется **дефорестацией (deforestation)**. В результате две функции, как говорят, “сплавляются” — fuse.

$$append\ (append\ xs\ ys)\ zs$$

transforms to

$$\begin{array}{l}
h_0\ xs\ ys\ zs \\
\textbf{where} \\
h_0\ xs\ ys\ zs \;=\; \textbf{case}\ xs\ \textbf{of} \\
\qquad\qquad Nil \;:\; h_1\ ys\ zs \\
\qquad\qquad Cons\ x\ xs \;:\; Cons\ x\ (h_0\ xs\ ys\ zs) \\
h_1\ ys\ zs \;=\; \textbf{case}\ ys\ \textbf{of} \\
\qquad\qquad Nil \;:\; zs \\
\qquad\qquad Cons\ y\ ys \;:\; Cons\ y\ (h_1\ ys\ zs)
\end{array}$$

$$flip\ (flip\ zt)$$

transforms to

$$\begin{array}{l}
h_0\ zt \\
\textbf{where} \\
h_0\ zt \;=\; \textbf{case}\ zt\ \textbf{of} \\
\qquad\qquad Leaf\ z \;:\; Leaf\ z \\
\qquad\qquad Branch\ xt\ yt \;:\; Branch\ (h_0\ xt)\ (h_0\ yt)
\end{array}$$

Рис. 15: Примеры работы дефорестирующего алгоритма Wadler [1988].

Термин и первый дефорестирующий алгоритм был предложен Philip Wadler Wadler [1988], он основан на нескольких простых правилах переписывания, нацеленных получить ситуацию `case K args of ...`, и агрессивном инлайнинге (см. примеры работы на рис. 15). Однако, этот алгоритм может приводить к экспоненциальному разбуханию кода и имеет шанс не завершится при наличии рекурсивных вызовов[84]. Чтобы алгоритм завершался программы должны быть написаны в некоторой строгой форме под названием treeless.

Сфокусируемся на списках и рассмотрим более практичные решения[85]. Первым очевидным решением было бы для каждой пары функций, работающих со списками, добавить специальное правило переписывания (см. 3.1.6), дефорестирующее с помощью алгебраических свойств списочных трансформаций. Однако, таких правил будет экспоненциально много.

```
1 {-# RULES
```

[84] Дефорестация является частным случаем суперкомпиляции Романенко, которая в свою очередь является обобщением большого количества компиляторных оптимизаций.

[85] `https://markkarpov.com/tutorial/ghc-optimization-and-fusion.html`

```
std::variant<Msg1, Msg2>
    deserialize(bytes bs) {
    if (...) {
        return std::variant{Msg1(...)};
    else {
        return std::variant{Msg2(...)};
    }
}
```

```
template<class Impl>
auto deserialize(bytes bs) {
    if (...) {
        return Impl::processMsg1(...);
    else {
        return Impl::processMsg2(...);
    }
}
```

Рис. 16: Ручная дефорестация в C++.

```
  "map/map" forall f g xs. map f (map g xs) = map (f . g) xs
    #-}
```

Современная техника дефорестирования в Haskell — fold/build list fusion Gill et al. [1993] — вместо использования множества алгебраических правил, определяет универсальный способ конструирования и деконструирования списка. Деконструировать список будем с помощью `foldr`. Конструировать будем с помощью функции `build`:

```
build :: (forall b . (a -> b -> b) -> b -> b) -> [a]
build g = g (:) []
```

Например, список `[1, 2, 3]` и функция `map` теперь запишутся следующим образом[86]:

```
list123 :: [Int]
list123 = build \s z -> s 1 (s 2 (s 3 z))

map :: (a -> b) -> [a] -> [b]
map f xs = build \s z -> foldr (\x acc -> s (f x) acc) z xs
```

Можно заметить, что на вход `build` передаётся список Чёрча, отсюда не удивителен закон:

```
foldr f ini (build g) ≡ g f ini
```

Таким образом, мы избавились от конструирования списков путём замены вызовов конструкторов на вызовы сворачивающих функций.

Дефорестацию также можно производить вручную (см. 16).

4.4.2 Visitor pattern

Рассмотрим некоторое дерево и его свёртку:

```
data Tree a = Leaf | Node a [Tree a]
foldTree :: Tree a -> r -> (a -> [r] -> r) -> r
```

[86] В стандартной библиотеке Haskell функции работы со списками написаны нормально, но рядом написаны правила `RULES` (см. 3.1.6), которые подменяют их реализацию на fold/build.

Перепишем и переименуем:

```
data Visitor a r = Visitor { onLeaf :: r, onNode :: a -> [r] -> r }
visitTree :: Tree a -> Visitor a r -> r
```

Чтобы это ещё более выглядело в ООП стиле, само дерево должно задаваться свёрткой (как бы интерфейсом с функцией `visit`), а разные вершины — конкретными её реализациями (объектами-наследниками):

```
data Tree a = Tree { visit :: forall r . Visitor a r -> r }

leaf :: Tree a
leaf = Tree { visit = \Visitor{onLeaf} -> onLeaf }

node :: a -> [Tree a] -> Tree a
node x ts = Tree { visit = \v@Visitor{onNode} -> onNode x (map (`visit` v) ts) }
```

Для наглядности, покажем этот код ещё и на Kotlin:

```
interface Visitor<a, r> {
    fun onLeaf(): r
    fun onNode(x: a, subtrees: List<r>): r
}

interface Tree<a> {
    fun <r> visit(visitor: Visitor<a, r>): r
}
class Leaf : Tree<Nothing> {
    override fun <r> visit(visitor: Visitor<Nothing, r>): r = visitor.onLeaf()
}
class Node<a>(val value: a, val subtrees: List<Tree<a>>) : Tree<a> {
    override fun <r> visit(visitor: Visitor<a, r>): r =
        visitor.onNode(value, subtrees.map { it -> it.visit(visitor) })
}
```

4.5 Всё через развёртку

Вспомним, что существует универсальная развёртка — анаморфизм, которая по генерирующей процедуре позволяет получить целую структуру данных (см. 4.3.3).

```
ana :: Functor f => forall s . (s -> f s) -> s -> Fix f
ana psi = In . fmap (ana psi) . psi
```

Приведём функцию `ana` к типу вида `A -> B`, чтобы затем проще было исследовать стрелку `B -> A`. Для этого сделаем `uncurry` и перенесём квантор налево от стрелки (он изменится на противоположный):

```haskell
ana :: Functor f => (exists s . (s, s -> f s)) -> Fix f
```

Закодируем квантор существования с помощью нового типа (см. 3.4.2) и перепишем анаморфизм на работу с ним:

```haskell
data Box f where
  --     exists s. (s,    s -> f s)
  Box :: forall s . s -> (s -> f s) -> Box

ana' :: Functor f => Box f -> Fix f
ana' (Box currSeed psi) =
  In $ (\nextSeed -> ana' (Box nextSeed psi)) <$> psi currSeed
```

Теперь построим изоморфизм между структурами данных и их тривиальными развёртками, возвращающими каждый раз следующий слой данной структуры, `Fix f` $\simeq$ `Box f`:

```haskell
to :: Fix f -> Box f
to x = Box x out

from :: Functor f => Box f -> Fix f
from = ana'
```

Таким образом, мы получили свидетельство того, что любую рекурсивную структуру данных можно хотя бы тривиальным образом представить как `Box f`. Иногда такое представление называют **co-Church encoding** Gibbons [2008].

Например, бесконечный ленивый список натуральных чисел может быть задан следующим образом. Заметьте, что тут мы *не полагаемся на ленивость Haskell*, а значит можем использовать эту технику и в энергичных языках.

```haskell
nats :: Box (ListF Int)
nats = Box 0 \curr -> Cons curr (curr + 1)
```

Упражнение 31 *Реализуйте ленивую функцию* `take :: Int -> Box (ListF Int) -> Box (ListF Int)`.

Иначе говоря, бесконечные структуры данных представимы как процедуры, лениво генерирующие слой рекурсивной структуры за слоем по запросу.

4.5.1 Абстрактные типы данных

Рассмотрим случай, когда функтор формы представляет собой произведение:

$$\texttt{s -> f s} \cong \left(\prod_i (t_i\ s)\right)^S \cong \prod_i (s \to t_i\ s)$$

То есть коалгебра эквивалентна кортежу функций. Таким образом, `Box` это не что иное, как *абстрактный тип данных (ADT)*: он включает в себя скрытое состояние неизвестной природы и набор операций для работы с ним Gibbons [2008]. На самом деле мы уже встречали такую конструкцию ранее, когда говорили про экзистенциальные типы 3.4.2 (там мы использовали инстансы классы типов в качестве кортежей функций).

ООП объекты тоже можно рассматривать через коалгебры. Подробнее про соотношение объектов и абстрактных типов данных можно почитать в Cook [2009].

4.5.2 Stream fusion

Ранее мы рассматривали foldr/build list fusion оптимизацию, элиминирующую промежуточные списки (см. 4.4.1). Однако, эта техника не подходит для многих популярных функций, например, `zip` (нужно корутинить между двумя алгебрами) или `take` (нужно оборвать свёртку в определённый момент).

Позднее была предложена техника использования ко-структуры списка — развёртки или `Stream`[87] Coutts et al. [2007]:

```haskell
data ListF a r = Nil | Cons a r
type MyStream a = Box (ListF a) -- ∃s . (s -> ListF a s, s)
-- ≅
data Step a s = Done | Yield a s
data Stream a where
  Stream :: forall s . (s -> Step a s) -> s -> Stream a

stream :: [a] -> Stream a
stream xs = Stream (\case [] -> Done; x:xs -> Yield x xs) xs

unstream :: Stream a -> [a]
unstream (Stream next s) = case next s of
  Done -> []
  Yield a s' -> a : unstream next s'
```

Идея в том, что теперь *функции работы со стримами нерекурсивны* и легко подвергаются базовым компиляторным дефорестирующим трансформациям в стиле Wadler [1988]. Действительно, вместо рекурсивного вызова мы запоминаем состояние для следующей итерации.

```haskell
mapS :: (a -> b) -> Stream a -> Stream b
mapS f (Stream next s) = Stream next' s
  where
    next' s = case next s of
      Done -> Done
```

[87] https://markkarpov.com/tutorial/ghc-optimization-and-fusion.html

```
      Yield x s' -> Yield (f x) s'
```

Общий пайплайн выглядит следующим образом:

1. Мы пишем все функции работы со списками через стримы:

```
map :: (a -> b) -> [a] -> [b]
map f = unstream . mapS f . stream
```

2. С помощью RULES (см. 3.1.6) задаём правило переписывания stream/unstream, двойственное foldr/build: `stream . unstream = id`.
3. Далее справляются обычные компиляторные оптимизации.[88]

Чтобы поддержать нерекурсивную функцию фильтрации, обычно добавляют отдельный вид шагов:

```
data Step a s = Done | Skip s | Yield a s

filterS :: (a -> Bool) -> Stream a -> Stream a
filterS p (Stream next s) = Stream next' s
  where
    next' s = case next s of
      Done -> Done
      Skip s' -> Skip s'
      Yield a s' -> if p a then Yield a s' else Skip s'
```

В стандартной библиотеке Haskell всё-таки используется foldr/build fusion, однако существует множество промышленных библиотек стримов [Bragilevsky, глава 14].

Больше про стримы можно почитать у Олега Киселёва[89].

4.6 Вездесущий дуализм

Мы рассмотрели три формы рекурсивных данных:

- `Fix f` — обычные рекурсивные типы данных, конечные в энергичных языках;
- `∀a . (f a -> a) -> a` — Church encoding, конечные структуры заданные свёртками;
- `∃s . (s -> f s, s)` — co-Church encoding, потенциально бесконечные структуры данных, абстракции по данным (работаем через интерфейс над сокрытым представлением).

В этом разделе мы рассмотрим некоторые практические примеры, в которых пространство решений задано дуальными представлениями коллекций.

[88] Можно гарантировать полную дефорестацию с помощью staging (см. далее **??**) Kiselyov et al. [2017].

[89] `https://okmij.org/ftp/Streams.html`

4.6.1 Push vs pull streaming

Рассмотрим различные представления списков или, как их иногда называют, потоков событий. Подробнее сравнение можно посмотреть тут [Kiselyov et al., 2017, параграф 3].

Church encoding для списков даёт **push stream**, или **internal iteration**, или **Observer pattern**[90] (в некотором смысле частный случай Visitor pattern 4.4.2), когда мы регистрируем обработчика событий, а коллекция вызывает (push) на нём методы. Примерами push потоков являются Java Stream API[91], Closure Transducers[92].

```
events.observe(object : Observer<Event> {
    override fun onComplete() { .. }
    override fun onNext(elem: Event) { .. }
})
```

co-Church encoding для списков даёт **pull stream**, или **external iteration**, или **Iterator pattern**. Чтобы получить следующий элемент, его нужно явно запросить (pull) потребляющему коду. Это представление часто гораздо удобнее с точки зрения использования (см. 4.5.2)[93], однако сложнее на стороне имплементации, потому что нужно выделить и поддерживать экзистенциальное состояние в коалгебре. Генераторы призваны автоматизировать выделение этого состояния, так как можно заметить, что это continuation вычисления (см. далее 6.4.3).

Нередко работу с потоками называют **реактивным программированием**[94].

4.6.2 Data vs codata

Вспомним функтор-формы для списка:

```
data ListF a r = Nil | Cons a r
```

С помощью ключевого слова `data` мы определяем новый алгебраический тип данных через способы его сконструировать. Далее, с помощью pattern-matching мы можем деконструировать значения алгебраического типа данных и воспользоваться хранящейся в нём информацией.

```
data List a where
  List :: ListF a (List a) -> List a

fold :: (ListF a r -> r) -> List a -> r
fold alg xs = case xs of
  Fix Nil -> alg Nil
  Fix (Cons y ys) -> alg (Cons y (fold alg ys))
```

[90] https://learn.microsoft.com/en-us/dotnet/api/system.iobserver-1?view=net-9.0
[91] https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html
[92] https://clojure.org/reference/transducers
[93] https://github.com/pulldown-cmark/pulldown-cmark/?tab=readme-ov-file#why-a-pull-parser
[94] (youtube) React 2014 : Erik Meijer - What does it mean to be Reactive?

Однако, также можно определять тип данных через способы его деконструировать с помощью ключевого слова `codata` (в воображаемом языке). Так, у ко-списка можно потребовать показать следующий слой структуры. Сконструировать такой тип данных можно с помощью co-pattern-matching'а, описав как деконструирующая функция должна вести себя на конструирующем терме (`unfold coalg s`)[95].

```
codata CoList a where
  force :: CoList a -> ListF a (CoList a)

unfold :: (s -> ListF a s) -> s -> CoList a
force (unfold coalg s) = case coalg s of
  Nil -> Nil
  Cons x s' -> Cons x (unfold coalg s')
```

По сути, `codata` представляет собой словарь функций, которые захватывают некоторое состояние, необходимое для генерации следующих слоёв структуры. Так, `codata` можно сравнить с ООП интерфейсами, а co-pattern-matching с анонимной реализацией интерфейса. Предыдущие два примера можно записать, например, в Kotlin следующим образом (форма списка кодируется как `Pair<a, r>?`, убедитесь самостоятельно, что этот тип изоморфен `ListF`).

```
class List<a>(val layer: Pair<a, List<a>>?)

fun <a, r> fold(alg: (Pair<a, r>?) -> r, xs: List<a>): r =
    if (xs.layer == null) alg(null)
    else alg(Pair(xs.layer.first, fold(alg, xs.list.second)))

interface CoList<a> { fun force(): Pair<a, CoList<a>>? }

fun <a, s> unfold(coalg: (s) -> Pair<a, s>?, ini: s): CoList<a> =
    object : CoList<a> {
        override fun force(): Pair<a, CoList<a>>? {
            val layer = coalg(ini)
            return if (layer == null) null else
                Pair(layer.first, unfold(coalg, layer.second))
        }
    }
```

Функциональный стиль программирование в основном оперирует напрямую наблюдаемыми данными `data`, а ООП — скрывает данные за интерфейсами `codata`. Однако, обе конструкции имеют свои понятные области применения и ими не стоит пренебрегать вне зависимости от предпочитаемого стиля Downen et al. [2019][96].

[95] (youtube) CS410 2017 Lecture 15 Coinduction and Coalgebras

[96] `https://reasonablypolymorphic.com/blog/review-codata/`

В языках с проверкой тотальности функций разделяют индуктивные и коиндуктивные определения. Для них используются паттерн-матчинг и ко-паттерн-матчинг, принцип индукции (катаморфизм) и принцип коиндукции (анаморфизм). Индуктивные определения задают конечные структуры данных, проверка тотальности проверяет, что рекурсивные вызовы делаются на структурно меньших подтермах. Коиндуктивные определения задают потенициально бесконечные структуры данных, для них проверяется продуктивность — порождающая функция всегда после рекурсивного вызова породит новый конструктор [97].

4.7 Приложение: категория алгебр

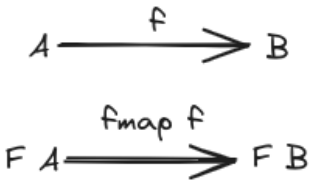
Это факультативный раздел, не являющийся необходимым для понимания курса в дальнейшем. Однако, полезен для понимания часто употребимой терминологии.

Категория — это коллекция объектов и коллекция стрелок. Для каждого объекта X существует тождественная стрелка, а для каждой пары стрелок существует способ получить их композицию: $f : Y \to Z, g : X \to Y \Rightarrow f \circ g : X \to Z$.

Определяют категорию, соответствующую Haskell — *Hask*. На самом деле это плохая категория с точки зрения теории, но для наших нестрогих рассуждений подойдёт[98]. Объектами в Hask являются типы языка Haskell, а морфизмами — термы, задающие функции между соответствующими типами. Тождественный морфизм — `id`, композиция задаётся как `f . g = \x -> f (g x)`.

Упражнение 32 *Как в такой категории представить константы?*

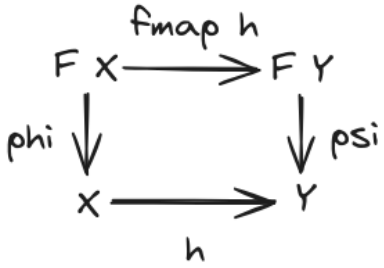
Функтором называется отображение между категориями, которое объектам одной категории сопоставляет объекты другой, а стрелкам одной — стрелки другой. В Haskell типовые конструкторы задают отображение между объектами, а `fmap` — между стрелками. Функтор должен сохранять тождественный морфизм и композицию.



Алгеброй в категории C называется пара из объекта категории $X \in Obj(C)$ и морфизма $\phi : F\ X \to X$, где F — функтор. Сам морфизм $F\ X \to X$ называют **f-алгеброй**. Алгебрами в смысле категорий можно описывать алгебры. Так, в качестве объекта X берём носитель алгебры. В качестве функтора F — сигнатуру алгебры в виде типа-формы. Тогда морфизмом будет интерпретацией сигнатуры.

[97] https://rocq-prover.org/doc/V8.18.0/refman/language/core/coinductive.html
[98] https://math.andrej.com/2016/08/06/hask-is-not-a-category/

Морфизмом алгебр называется такой морфизм между носителями $h : X \to Y$, что следующая диаграмма коммутирует. Говорят, что **диаграмма коммутирует**, если все возможные пути по стрелкам в ней равны.



В морфизме алгебр можно обнаружить знакомые черты гомоморфизмов, то есть операций между носителями, которые "уважают" операции сигнатуры алгебраической теории.

Алгебры над категорией C образуют **категорию алгебр**, в которой объектами являются алгебры, а морфизмами — морфизмы алгебр.

Начальным (инициальным) объектом категории называется объект, из которого в каждый другой объект существует уникальная стрелка. **Терминальным (финальным) объектом** категории называется объект, в который из каждого другого объекта категории существует уникальная стрелка.

Инициальный и терминальный объекты категории не обязательно присутствуют в единственном экземпляре. Но все инициальные объекты изоморфны друг другу, как и все терминальные.

Упражнение 33 *Приведите начальный и терминальный объекты категории Hask.*

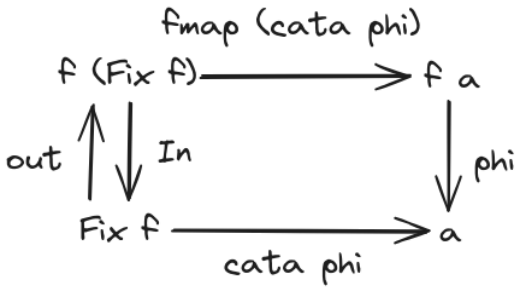
Рекурсивный тип — это тип, значит ему соответствует объект в категории Hask. X является рекурсивным типом с формой F, если имеет место следующий изоморфизм:

$$X \simeq F\ X$$

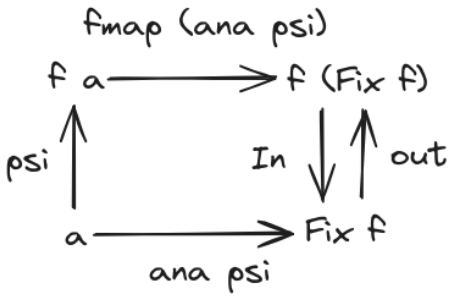
Можно заметить, что свидетель изоморфизма справа налево напоминает f-алгебру, а слева направо — f-коалгебру (всё то же самое, только все стрелки в обратную сторону). И действительно, подходящий объект X должен быть либо начальным объектом категории алгебр, либо терминальным объектом категории коалгебр (с соответствующими морфизмами). Первый вариант соответствует конечным структурам данных, второй — потенциально бесконечным.

Начальным объектом категории алгебр над *Hask* для функтора f является следующая алгебра: `(Fix f, In)`[99][100]. Действительно, для каждого типа `a` и для каждой f-алгебры `phi` мы можем построить такой морфизм `cata phi :: Fix f -> a`, что следующая диаграмма будет коммутировать:

[99] https://bartoszmilewski.com/2017/02/28/f-algebras/
[100] https://ncatlab.org/nlab/show/catamorphism



Покажем, что `(Fix f, out)` является терминальным объектом категории коалгебр (благодаря ленивости Haskell). Аналогично, для каждого объекта `a` и f-коалгебры `psi` найдётся морфизм `ana psi :: a -> Fix f`:



5 Интерпретаторы

Искусство программирования во многом состоит в умении управлять сложностью, и в рамках данного курса основным инструментом для этого мы будем рассматривать построение языков и интерпретаторов.

5.1 Интерпретаторы как основа основ

Мы начнём с обзора роли интерпретаторов в программировании.

5.1.1 Башня интерпретаторов

Самым базовым интерпретатором является процессор, он воплощён физически в железе. Ему на вход подаётся программа на некотором языке, например, x86, он зачитывает команды и превращает их в действия над памятью. Однако, человеку сложно программировать на этом языке, нужен новый язык, инкапсулирующий часть сложности и скрывающий лишние детали.

Чтобы получить новый язык, мы строим программный интерпретатор. **Программный интерпретатор** U_M^N — это программа на языке M[101], получающая на вход программу на языке N и вход для неё — данные из D, и возвращающая результат выполнения этой программы на этих данных:

$$U_M^N : N \times D \to D$$

Язык реализации интерпретатора M мы будем называть **мета-языком**, а L — **определяемым**. Про интерпретатор можно интуитивно думать следующим образом: это понятное мета-языку объяснение того, что значат конструкции определяемого языка. Иными словами, какие инструкции мета-языка нужно исполнить, чтобы получить нужную семантику инструкций определяемого языка.

Например, у нас есть программа p_N и данные для неё d_{in}, результат исполнения этой программы d_{out} можно получить как

$$d_{out} = U_M^N \left(\underbrace{\langle p_N, d_{in} \rangle}_{\in N \times D} \right)$$

Но интерпретатор это тоже программа. Как её запустить? Возьмём наш базовый интерпретатор U^{x86}, у него нет языка реализации, так как он реализован в железе, а не программно. Возьмём интерпретатор языка ассемблера, реализованный в кодах x86, U_{x86}^{Asm}, программу на ассемблере p_{Asm} и вход для неё d_{in}. Вспомним, что программа — это тоже данные, просто в некотором специальном формате. Тогда результат применения p_{Asm} на данных мы получим

[101] Под языком мы тут понимаем множество программ на этом языке, иначе говоря, множество деревьев определённого вида.

следующим образом:

$$d_{out} = U^{x86}\left(\left\langle \underbrace{U^{Asm}_{x86}}_{\in x86}, \underbrace{\overbrace{\langle p_{Asm}}^{\in Asm}, \overbrace{d_{in}\rangle}^{\in D}}_{\in D} \right\rangle\right)$$

Но язык ассемблера, тоже не очень приятен для программирования. Однако, на нем можно уже написать интерпретатор языка посложнее. И так далее. Получаем *башню интерпретаторов*, на вершине находится язык, на котором мы хотим уже решать непосредственно нашу задачу:

$$d_{out} = U^{x86}\left(\left\langle U^{Asm}_{x86}, \left\langle U^{C}_{Asm}, \left\langle U^{Has}_{C}, \langle p_{Has}, d_{in}\rangle\right\rangle\right\rangle\right\rangle\right)$$

На практике часто язык задают через трансляцию (компиляцию) в другой. Однако, далее мы будем фокусироваться на встроенных языках и понятие интерпретатора будет нам полезнее. Кроме того, существуют универсальные теоретические способы построения компилятора по интерпретатору[102] (см. далее **??**), так что для нас они схожи.

5.1.2 Интерпретаторы повсюду

Хорошо, мы пришли к языку нашего сердца (Хаскеллу), почему же мы продолжаем говорить об интерпретаторах? Потому что для решения конкретных бизнес-задач прикладные языки всё ещё слишком церемониальны — программисту приходится думать о большом количестве вещей, нерелевантных его предметной области и решаемой задаче. Сложность — главный враг программиста, потому что ресурсы человеческого мозга несопоставимы со сложностью реальности, которую приходится описываться в программах. Таким образом, в работе постоянно приходится описывать новые языки, наиболее подходящие для решения конкретных прикладных задач. А новые языки мы задаём с помощью интерпретаторов.

Как выглядит классический рекурсивный интерпретатор? Он получает программу в виде некоторого дерева и рекурсивно обходит его, считая результаты поддеревьев. Когда он посещает вершину дерева, он определяет её тип и понимает, какие действия нужно исполнить. То есть тип вершины диспатчит, навигирует, исполнение интерпретатора на нужный код. Так, интерпретатор простой языка выражений имеет следующий вид:

```haskell
eval :: Expr -> Int
eval prog = case prog of
  Const x -> x
  Plus l r -> eval l + eval r
```

Видно, что это похоже, например, на работу утилиты командной строки — разбираем аргументы, определяем, что и как нужно сделать, делаем. Как ни странно, философия Unix, в частности, заключается в построении маленьких языков (утилит с текстовым API), решающих хорошо одну задачу Bentley [1986]. Ещё это похоже на обработку запроса web-сервером — определяем роут на которую пришел запрос, выполняем соответствующее действие. То есть

[102] https://en.wikipedia.org/wiki/Partial_evaluation

не так редко мы в реальной жизни пишем интерпретаторы. Мы просто не видим, что то, что мы пишем — это на самом деле интерпретатор некоторого языка. В общем случае, свёртку структуры данных уже можно рассматривать как интерпретацию Gibbons and Wu [2014].

Более того, как мы убедимся в дальнейшем 5.4, написание любой функции — это уже задание нового языка. Вот был язык, в котором нельзя было добавить пользователя в приложение. Написали функцию `registerUser` — появилась новая команда в языке — добавить пользователя. Далее мы формально покажем, то такой способ эквивалентен добавлению новой ноды в синтаксическое дерево языка (5.4.1). Использование функций является примером встраивания языка, когда мы вместо того, чтобы делать новый отдельный язык, его реализуем как библиотеку для уже существующего языка Gibbons [2013].

Как мы убедимся в течение курса, многие задачи можно рассматривать как придумывание языка и реализацию интерпретатора. Значит, если мы научимся хорошо писать интерпретаторы, мы научимся делать сразу кучу всего! И основные наши усилия будут направлены на изучение средств построения интерпретаторов встроенных языков.

Во время повествования мы часто пользуемся приёмом Hutton's Razor, который подразумевает рассмотрение до смешного маленького языка для изучения сложных концепций. Утверждается, что для изучения большинства вопросов можно сконструировать такой язык, делающий всё важное максимально наглядным.

5.1.3 Интерпретаторы и семантика языков программирования

Семантика языков программирования[103] — это наука, изучающая свойства языков и смысл программ, его свойства и способы описания. Отличным введением может послужить серия книг Software Foundations[104] Pierce et al. [2010]. Существует много различных стилей описания семантики программ, для нас важнейшим будет денотационная семантика.

Денотационная семантика[105][106][107] описывает смысл программ путём сопоставления им объектов некоторого множества, **семантического домена**. Иначе говоря, денотационная семантика языка L — это тотальная функция из программы на этом языке в элемент домена D:

$$[\![\bullet]\!] : L \mapsto D$$

Домен выбирается исходя из языка и информации, которую хочется извлекать из программ. Например, чтобы узнать размер программы (тут, выражения со сложением), в качестве домена можно взять натуральные числа:

$$\begin{aligned} [\![n]\!] &= 1 \\ [\![l + r]\!] &= 1 + \max([\![l]\!], [\![r]\!]) \end{aligned}$$

[103] `https://en.wikipedia.org/wiki/Semantics_(computer_science)`
[104] `https://coq.vercel.app/ext/sf/`
[105] `https://en.wikipedia.org/wiki/Denotational_semantics`
[106] `https://en.wikibooks.org/wiki/Haskell/Denotational_semantics`
[107] (youtube) The Lost Art of Denotational Semantics — Eric Meyer.

Если нас интересует конечный результат, можно посчитать его:

$$\begin{array}{lcl} [\![n]\!] & = & n \\ [\![l + r]\!] & = & [\![l]\!] + [\![r]\!] \end{array}$$

Если у программы есть вход, доменом будет функция $\mathbb{N} \to \mathbb{N}$:

$$\begin{array}{lcl} [\![n]\!](m) & = & n \\ [\![l + r]\!](m) & = & [\![l]\!](m) + [\![r]\!](m) \\ [\![input]\!](m) & = & m \end{array}$$

Таким образом, *программа является лишь синтаксической записью для некоторого элемента семантического домена*. Вариантов доменов много, это могут быть даже игры[108].

Упражнение 34 *В какой домен разумно проинтерпретировать программы на языке с целочисленными мутабельными переменными? А на недетерминированном языке?*

Легко заметить, что денотационная семантика языка — это просто интерпретатор, только написанный на языке математики. Такие интерпретаторы ставят в основу башни интерпретаторов, когда цель исследовать свойства языков и программ, а не исполнять их.

Также, можем реализовывать интерпретатор на каком-нибудь реальном языке и он тоже будет задавать семантику определяемого языка. Однако формальность такого определения будет зависеть от формальности описания семантики мета-языка. Такие интерпретаторы называют **определяющими**, они задают семантику языка, как правило, жертвуя эффективностью ради наглядности. Взаимоотношения определяемого языка и мета-языка изучаются в классических статьях Reynolds [1972, 1998][109].

Мы будем использовать определяющие интерпретаторы для задания семантики новых языков и в качестве мета-языка будем использовать Haskell. А в качестве доменов будем брать типы Haskell. И интерпретировать программу не в множество функций между натуральными числами, а, скажем, в тип `Nat -> Nat` в языке Haskell[110]. Так, денотационная семантика языка сумм с входом будет записываться следующим образом:

```
eval :: Prog -> (Nat -> Nat)
eval = \case
  Val n -> \_ -> n
  Plus l r -> \m -> eval l m + eval r m
  Input -> \m -> m
```

Семантика называется **композиционной (compositional)**, если смысл конструкций зависит только от смысла подконструкций. Иначе говоря, если денотационная семантика представляет собой свёртку программы (рис. 17) и может быть записана в терминах катаморфизма 4.3.3.

[108] https://en.wikipedia.org/wiki/Game_semantics

[109] Активно используемое автором понятие продолжения будет рассмотрено далее в этом курсе (раздел 6).

[110] https://okmij.org/ftp/Denotational.html

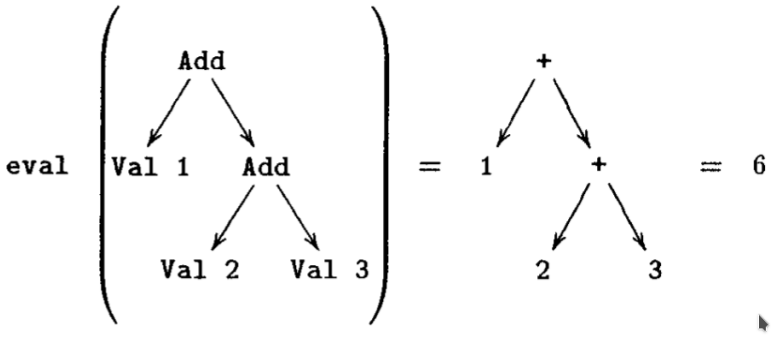


Рис. 17: Денотационная семантика определяет смысл синтаксических конструкций через операции над доменом Hutton [1998].

Другим популярным стилем описания семантики является **операционная семантика**, которая представляет смысл программы в виде последовательности шагов вычислений. Это может быть как последовательное переписывание самого выражения, так и переписывание состояния некоторой абстрактной машины. Операционная семантика, в свою очередь, задаётся как развёртка (или анаморфизм 4.3.3) последовательности шагов вычисления из программы. Тут отчётливо видна некоторая двойственность между денотационной и операционной семантиками Hutton [1998].

5.1.4 Встроенные доменно-специфичные языки (eDSL)

Под **доменно-специфичными языками (domain specific languages, DSL)**[111] часто понимают специализированные языки для конкретных предметных областей, например, запросов к БД или форматирования документов. Как правило, такие языки не являются полными по Тьюрингу.

В этом курсе, однако, мы будем считать доменно-специфичным языком любую доменно-специфичную специализацию языка общего назначения[112]. Это следует из того соображения, что код должен читаться как грамотная проза с уместным словоупотреблением, предоставляющая читателю только необходимое количество подробностей, скрывая несущественное за умолчаниями и терминологией.

Самостоятельные доменно-специфичные языки (standalone domain specific languages) — языки, имеющие свой собственный конкретный синтаксис, а так же инструменты программирования (IDE, исполняющая среда. . .). Примеры: SQL, AWK, Antlr. . . [113]

Встроенные доменно-специфичные языки (embedded domain specific languages, eDSL)

[111] `https://en.wikipedia.org/wiki/Domain-specific_language`

[112] `https://en.wikipedia.org/wiki/Language-oriented_programming`, на русскоязычную страницу тоже следует заглянуть.

[113] Есть отличная прикладная книга Nystrom, рассказывающая о построении интерпретаторов и простых виртуальных машин. В то же время она покрывает сознание полноценного языка во всех его аспектах, от синтаксического анализа до управления памятью.

— языки, пользующиеся поддержкой инфраструктуры других языков. Обычно реализуются как библиотеки для программ на уже существующем языке общего назначения. Не имеют полностью собственного конкретного синтаксиса. Примеры: ORM, функции обработки строк, библиотека парсер-комбинаторов. . .

Deep eDSL — термы на таком языке строят дерево абстрактного синтаксиса для дальнейшей интерпретации:

```haskell
f :: Int -> Int
f = eval $ Const 1 `Plus` Input
```

Интерпретаторы, которые принимают деревья на вход и интерпретируют их в семантический домен, называют **инициальными**[114], мы их также будем называть "классическими". С них мы и будем начинать. Классические интерпретаторы полезны, например, как для реализации "последнего языка" — интерфейса программы во внешний мир, и как фундамент для наших дальнейших построений.

Однако, можно заметить, что промежуточное дерево, которое получается, нас, как правило, не интересует. Нам важно только получить элемент домена, которым мы уже умеем пользоваться непосредственно. **Shallow eDSL** минуют стадию построения дерева и сразу строят значение в семантическом домене. Такие интерпретаторы мы будем называть **финальными**.

```haskell
cnst :: Int -> (Int -> Int)
cnst x _ = x

input :: Int -> Int
input env = env

plus :: (Int -> Int) -> (Int -> Int) -> (Int -> Int)
plus l r env = l env + r env

f :: Int -> Int
f = cnst 1 `plus` input
```

Интерпретаторы часто называют **наблюдателями (observers)**, которые анализируют термы и дают им некоторый смысл Gibbons [2013]. Можно заметить, что для deep eDSL можно написать сколь угодно много различных наблюдателей. Однако в случае shallow embedding наблюдатели всегда `id`. Мы будем обсуждать возможные решения этой проблемы далее в разделе 5.4.1.

Обсуждение терминологии и сравнение подходов к построению DSL можно найти в Gibbons [2013]. Краткое описание терминов — в конспекте курса Language Engineering Hutchinson.

Введём ещё одно важное понятие. **Meta-circular интерпретатор**[115] — это интерпретатор, определяющий конструкции определяемого языка через конструкции мета-языка Reynolds [1972]. Например:

[114] Тип синтаксиса заданный с помощью `data` является инициальным объектом категории интерпретаций.

[115] `https://en.wikipedia.org/wiki/Meta-circular_evaluator`

```haskell
interpret term = case term of
  App f t -> (interpret f) (interpret t)
  If c t e -> if interpret c then interpret t else interpret e
  ...
```

Свойства мета-языка в таком случае во многом определяют свойства объектного Reynolds [1972, 1998]. Мы будем в этом курсе стремиться как можно более переиспользовать возможности мета-языка.

Упражнение 35 *Предположите, какие свойства наследует определяемый язык из примера выше.*

5.1.5 Пример: библиотека Accelerate

Интересным примером встроенного языка, находящегося где-то между deep и shallow является библиотека Accelerate[116] [Marlow, 2011, глава 6]. Она позволяет на Haskell описать вычисления, которые будут исполняться на GPU[117].

Чтобы исполнить что-то на GPU, нужно породить и скомпилировать код на Cuda. Таким образом, язык Accelerate должен быть deep embedding, чтобы иметь дерево вычисления для трансляции в Cuda наиболее эффективным образом.

В то же время описывать численные вычисления как дерево крайне неудобно. Неплохо было бы иметь привычные операторы и функции высших порядков для работы с массивами на GPU. Поэтому Accelerate предоставляет на самом деле shallow интерфейс для построения деревьев. Так, для деревьев выражений определена реализация численных классов типов, например, `Num`, где операции просто достраивают дерево:

```haskell
example :: Acc (Vector Int) -> Acc (Vector Int) -> Acc (Vector Int)
example xs ys = A.zipWith (+) xs ys
```

5.2 Типы значений

Рассмотрим, какие есть способы реализации языков, значения в которых могут иметь различные типы.

5.2.1 Untyped tagless interpreters

Для начала рассмотрим некоторый тривиальный нетипизироватный язык. Под нетипизированностью понимаем отсутствие проверки типов как до исполнения программы, так и во время. Абстрактный синтаксис этого языка зададим следующим образом:

[116] `https://hackage.haskell.org/package/accelerate`

[117] Другой подход: Java code reflection, чтобы в рантайме извлекать модель кода. Однако, такой подход не предоставляет статически гарантий программисту и требует глубогоко внедрения в мета-язык.

```haskell
data Expr = Const Int | IsZero Expr | If Expr Expr Expr
```

Значения, возникающие во время исполнения программ на этом языке будем представлять значениями типов `Bool` и `Int` языка Haskell. Соответственно, семантическим доменом программы на этом языке является либо `Bool`, либо `Int`, в зависимости от самой программы.

```haskell
evalUnsafe :: Expr -> forall res . res
evalUnsafe = \case
  Const val -> unsafeCoerce val
  IsZero cond -> unsafeCoerce $ evalUnsafe @Int cond == 0
  If c t e -> if evalUnsafe c then evalUnsafe t else evalUnsafe e
```

Здесь `unsafeCoerce` используется, чтобы обмануть статическую систему типов Haskell и просто исполнять программы на нашем нетипизированном языке. Мы имеем право так делать, поскольку `Int` и `Bool` в Haskell имеют одинаковый размер. Неверное написание программы на этом языке или выбор неправильного домена интерпретации приводят к падению.

5.2.2 Typed tagged interpreters

Чтобы добиться некоторой безопасности исполнения, будем приписывать значениям теги, которые будут доступны во время исполнения. Заведём следующий алгебраический тип:

```haskell
data RtValue = RtBool Bool | RtInt Int
```

Теперь семантическим доменов у нас будет тип `RtValue`, а интерпретатор сможет проверять типы во время исполнения:

```haskell
evalRt :: Expr -> RtValue
evalRt (IsZero expr) = case evalRt expr of
  RtBool value -> error "Type error"
  RtInt value -> RtBool (value == 0)
-- ...
```

Ситуация с безопасностью программы определённо стала лучше, однако проверка типов во время исполнения — это уже поздно: требует дополнительных расходов производительности и удорожает тестирование.

Этот подход часто называют **динамической типизацей**, когда мы атрибутируем значения некоторой типовой информацией для использования во время исполнения (см. также 3.4.6).

5.2.3 Typed tagless interpreters

Опишем систему типов нашего маленького языка.

$$\frac{n : Int}{Const\ n : int}\ Const \qquad \frac{n : int}{IsZero\ n : bool}\ IsZero \qquad \frac{c : bool \quad t : \tau \quad e : \tau}{If\ c\ t\ e : \tau}\ If$$

Можем в качестве типовых тегов переиспользовать типы Haskell:

$$int \rightsquigarrow Int, bool \rightsquigarrow Bool$$

Заметим, что с помощью обобщённого алгебраического типа данных `Expr ty` (2.1.4), мы как раз закодировали эти правила вывода. Иначе говоря, мы получили статически типизированный язык программирования, переиспользовав систему типов Haskell.

```haskell
data Expr ty where
  Const :: Int -> Expr Int
  IsZero :: Expr Int -> Expr Bool
  If :: forall ty . Expr Bool -> Expr ty -> Expr ty -> Expr ty

eval :: Expr ty -> ty
eval = \case
  Const n -> n
  IsZero e -> eval e == 0
  If c t e -> if eval c then eval t else eval e
```

Благодаря статической типизации, мы можем отказаться от тегирования значений во время интерпретации без потери безопасности.

5.3 Связывания и функции первого класса

В этом параграфе мы рассмотрим техники и понятия, относящиеся к реализации функций первого класса и связываний в общем. Поскольку эта функциональность уже, как правило, реализована в мета-языке, мы будем стремиться её максимально переиспользовать.

`let`-связывания можно представить через функции первого класса следующим образом:

$$\textbf{let } x := N \textbf{ in } M \equiv (\lambda x.\, M)\ N$$

Напомним, что от функций первого класса можно избавиться с помощью дефункционализации, рассмотренной ранее 3.1.7.

5.3.1 Семантика имён

Существует несколько способов задания семантики идентификаторам.

Динамическое связывание (dynamic scoping) — значение свободных переменных функции зависит от области видимости в месте вызова. То есть разрешение имени происходит в момент обращения к переменной. Например, следующий код напечатает 42:

```scala
val x = 4
val f = () => x + 1
val x = 41
println(f())
```

Этот подход проще в реализации и использовался в ранних версиях Lisp'ов, например. Однако в таком случае функции не являются надежным барьером абстракции, по-хорошему все свободные переменные должны являться частью сигнатуры (вернёмся в этому в главе про системы эффектов **??**).

Лексическое/статическое связывание (lexical/static scoping) — переменные связываются со значениями в момент объявления функции, в момент вызова результат зависит только от параметров (по модулю изменяемого состояния[118]). Слово "лексический" часто употребляется в языках, когда мы что-то можем понять из исходного кода без запуска программы. Так, код из примера выше напечатает 5.

Далее в этом разделе мы будем говорить о различных способах реализации функций первого класса со статическим связыванием переменных, которых, на самом деле, великое множество[119].

5.3.2 Подстановки

Как можно заметить, в классическом лямбда-исчислении подстановки от бета-редукции (вспоминали в разделе 1.1) обеспечивают статическое связывание. Действительно, аргумент немедленно подставляется во все вхождения переменной, соответственно она не остаётся свободной, а просто исчезает.

$$(\lambda x.\,(\lambda x.\,\lambda y.\,x + y)\ 4)\ 41 \rightsquigarrow (\lambda x.\,(\lambda y.\,4 + y))\ 41$$

Такой подход не является самым эффективным, потому что на каждую аппликацию требуется переписывать код функции целиком (!). В то же время его довольно просто реализовать для некоторых представлений лямбда-термов. Рассмотрим пример такого представления — **locally nameless** Charguéraud [2012].

```haskell
data Term var
  = Var var
  | App (Term var) (Term var)
  | Lam (Term (Maybe var))
```

В этом представлении можно выбирать любой тип для именования свободных переменных:

```haskell
example :: Term String
example = Var "x" `App` Var "y" -- x y
```

Добавление каждой связанной переменной добавляет типу переменных нового обитателя `Nothing` для обращения к ближайшей связанной переменной:

[118] Например, в Kotlin в лямбды можно захватывать изменяемые переменные. Изменения снаружи наблюдаемы внутри лямбды, и наоборот. Иногда это может быть очень удобно, однако нередко приводит к очень неочевидному поведению.

[119] `https://jesper.cx/posts/1001-syntax-representations.html`

```haskell
-- λx.x y
example1 = Lam $ Nothing `App` Just "y"
-- λx y.x y z
example2 = Lam $ Lam $ Just Nothing `App` Nothing `App` Just (Just "z")
```

Монадический bind является реализацией подстановки для таких термов:

```haskell
instance Monad Term where
  (>>=) :: Term var -> (var -> Term var') -> Term var'
  Var var >>= subst = subst var
  App l r >>= subst = App (l >>= subst) (r >>= subst)
  Lam t >>= subst = Lam $ t >>= \case
    Nothing  -> Var Nothing
    Just var -> Just <$> subst var
```

Упражнение 36 *Подумайте, зачем нужен `fmap Just` в последней строчке.*

Соответственно, call-by-name интерпретатор такого лямбда-исчисления будет выглядеть следующим образом:[120]

```haskell
eval :: Term var -> Term var
eval = \case
  Var var -> Var var
  App f arg -> case eval f of
    Lam body -> eval $ body >>= maybe arg Var
    t -> App t (eval arg)
  Lam t -> Lam (eval t)
```

5.3.3 Окружение

Можно делать подстановку значений переменных лениво, распространяя окружение, которое ставит в соответствие свободным переменным термы. Лучше, в целом, пока не стало, но мы получили композиционную семантику из некомпозиционной путём эксплицирования контекстных зависимостей (подробнее далее 5.4.3).

```haskell
data Term1 = Var1 String | App1 Term1 Term1 | Lam1 String Term1
type Env = Map String Term1

eval1 :: Term1 -> Env -> Term1
eval1 term env = case term of
  Var1 name -> Map.findWithDefault (Var1 name) name env
```

[120] Поскольку мы рассматриваем классическое λ-исчисление, в качестве результирующего значения мы получаем тоже терм, но в нормальной форме.

```
6     App1 f arg -> case eval1 f env of
7       Lam1 name body -> eval1 body (Map.insert name arg env)
8       t -> App1 t (eval1 arg env)
9     Lam1 name body ->
10      let env' = Map.delete name env in
11      Lam1 name (eval1 body env')
```

Упражнение 37 *Объясните, зачем окружение модифицируется на строчке 10?*

Если ветка Lam1 не будет рекурсивно обходить подтерм и подставлять значения переменных, информация о значениях свободных переменных в нём потеряется и мы получим динамическое связывание вместо статического. Чтобы восстановить статическое связывание, ветка Lam1 интерпретатора должна конструировать замыкание, включающее текущее окружение (см. далее 5.3.4).

5.3.4 Замыкания

Чтобы не делать энергично подстановку в тела функций и сохранить при этом статическое связывание, добавим ещё одну конструкцию, **замыкание (closure)**[121][122] [Nystrom, глава 11]. Оно будет хранить контекст, в котором должен исполняться соответствующий терм.

```
data Term1 = Var1 String | App1 Term1 Term1 | Lam1 String Term1
           | Closure Env String Term1 -- только для вычислений
type Env = Map String Term1

eval1 :: Term1 -> Env -> Term1
eval1 term env = case term of
  Var1 name -> Map.findWithDefault (Var1 name) name env
  App1 f arg -> case eval1 f env of
    Closure env' body ->
        let arg' = eval1 arg env in
        eval1 body (Map.insert name arg' env')
    t -> App1 t (eval1 arg env)
  Lam1 name body -> Closure env name body
```

Замыкания обычно и используют в промышленных языках как представление времени исполнения функций высших порядков. Во время компиляции сначала производят **closure conversion** — функции высших порядков представляют как пару из окружения и указателя на функцию, принимающую окружение дополнительным аргументом. Теперь, когда функция не содержит свободных переменных, делают **lambda lifting**[123] — поднимают её на верхний уровень. Подробные примеры можно посмотреть в гарвардских слайдах Chong.

[121] https://en.wikipedia.org/wiki/Closure_(computer_programming)
[122] Термин closure был предложен Piter Landin, вместе с кучей других вещей.
[123] https://en.wikipedia.org/wiki/Lambda_lifting

5.3.5 Типизированный контекст

Рассмотрим кодирование, описанное, например, в Kiselyov [2012a].

Для начала научимся с помощью системы типов Haskell проверять валидность обращения к окружению. Представим окружение как список типов, закодированный с помощью вложенных пар:

```
(4, (4.0, "hello")) :: (Int, (Double, String))
```

Обращение к окружению будем кодировать числом в унарной записи. Тип числа (типизированной ссылки внутрь контекста) пусть задаёт множество окружений `env`, из которых на данной позиции можно извлечь тип `ty`.

```
data Ref env ty where
  Here :: Ref (ty, env) ty
  There :: Ref env ty -> Ref (ty', env) ty
```

Например, тип числа 1 утверждает, что с его помощью можно извлечь значение типа `ty` из контекста, в котором значение соответствующего типа находится на первой позиции (нумерация с нуля):

```
There Here :: Ref (ty', (ty, env)) ty
```

Теперь мы можем закодировать типизированное безопасное обращение к контексту:

```
envLookup :: env -> Ref env ty -> ty
envLookup env ref = case (ref, env) of
  (Here, (x, _)) -> x
  (There ref', (_, env')) -> envLookup env' ref'
```

Упражнение 38 *Можно ли разобрать пару сразу на строчке 2? Поясните.*

5.3.6 Meta-circular интерпретация

Крайне не хотелось бы для eDSL самостоятельно реализовывать связывания и функции первого класса. Построим meta-circular интерпретатор (см. 5.1.4), который будет переиспользовать функции первого класса мета-языка для реализации их в определяемом языке.

Термы теперь будут не только аннотированы результирующими типами, но и типами необходимых для интерпретации окружений, рассмотренных ранее 5.3.5. Абстрагированному терму доступно большее окружение.

```
data Term2 env ty where
  Var2 :: Ref env ty -> Term2 env ty
  App2 :: Term2 env (arg -> res) -> Term2 env arg -> Term2 env res
  Lam2 :: Term2 (arg, env) res -> Term2 env (arg -> res)
```

Теперь абстракцию можем проинтерпретировать в функцию Haskell, а аппликацию — в аппликацию:

```
eval2 :: Term2 env ty -> env -> ty
eval2 term env = case term of
  Var2 ref -> env `envLookup` ref
  App2 f arg -> (eval2 f env) (eval2 arg env)
  Lam2 t -> \arg -> eval2 t (arg, env)
```

Упражнение 39 *Как так получилось, что в последней строчке нужно принять ещё один аргумент?*

Упражнение 40 *Это call-by-value интерпретатор или call-by-name? От чего это зависит?*

Упражнение 41 *Подумайте, какое решение должно быть более производительно, это или предыдущее?*

Обратите внимание, что теперь функции определяемого языка во время исполнения — это просто функции мета-языка. А значит, в программах на определяемом языке мы *можем полностью переиспользовать мета-язык*! Добавим для этого конструкцию, позволяющую сохранить произвольное значение мета-языка в дереве:

```
data Term2 env ty where
  Val2 :: ty -> Term2 env ty
  -- ...

eval2 :: Term2 env ty -> env -> ty
eval2 term env = case term of
  Val2 x -> x
  -- ...

example :: Term2 env (Int -> Int)
example = Lam (Val2 (+) `App2` Val2 1 `App2` Var2 Here)
```

5.3.7 Синтаксис высшего порядка

Ещё чем мы ещё занимаемся вручную — определяем связыватели (да ещё и в унарной записи). Хотим переиспользовать их из мета-языка. Для это мы будем прямо в дереве синтаксиса хранить функции мета-языка — использовать **синтаксис высшего порядка (higher order abstract syntax)**[124][125] Pfenning and Elliott [1988]:

[124] https://en.wikipedia.org/wiki/Higher-order_abstract_syntax

[125] What is higher-order in higher-order abstract syntax?

```haskell
data Term3 ty where
  Val3 :: ty -> Term3 ty
  Plus :: Term3 Int -> Term3 Int -> Term3 Int
  App3 :: Term3 (arg -> res) -> Term3 arg -> Term3 res
  Lam3 :: (Term3 arg -> Term3 res) -> Term3 (arg -> res)

example3 :: Term3 Int
example3 = (Lam3 \x -> x `Plus` Val3 41) `App3` Val3 1
```

Интерпретация очень простая и абсолютно meta-circular:

```haskell
eval3 :: Term3 ty -> ty
eval3 term = case term of
  Val3 x -> x
  Plus l r -> eval3 l + eval3 r
  App3 f arg -> (eval3 f) (eval3 arg)
  Lam3 f -> \arg -> eval3 (f (Val3 arg))
```

Упражнение 42 *Можно ли было объявить* `Lam3` *следующим образом?*

```haskell
Lam3 :: (arg -> Term3 res) -> Term3 (arg -> res)
```

5.3.8 Сериализация функций

В этом разделе мы говорили о возможных реализациях функций первого класса, то есть функций, которые можно использовать так же гибко, как и данные. Возникает закономерный вопрос: можем ли мы сериализовать функцию первого класса и послать исполняться на другую машину?

Функция состоит из кода и захваченных свободных переменных в случае статического связывания. Соответственно, если код представлен в сериализуемом виде (например, позиционно-независимый байт-код), то его в принципе можно переслать по сети и исполнить на другом инстансе виртуальной машины. Так, например, делает Erlang. Однако, такой подход неэффективный, так как байт-код нужно интерпретировать или предварительно компилировать. Таким образом, Erlang жертвует скоростью исполнения ради горизонтальной масштабируемости.

Если мы гарантируем, что на различных узлах кластера исполняется один и тот же код, как обычно и бывает на практике, можно добиться более эффективной реализации. Например, используя дефункционализацию (см. 3.1.7), мы можем сериализовать только объекты алгебраического типа, кодирующие функции. Поскольку на другом узле кластера исполняется такой же код, мы там можем десериализовать объект и исполнить его с помощью `apply`. Однако этот подход не очень поддерживает модульность (сложно один алгебраический тип разбить на много), а так же `apply` каждый раз производит декодирование перед исполнением кода (чем, в прочем, можно пренебречь, учитывая работу с сетью).

Подход, реализованный в Haskell[126][127] позволяет наделить каждую функцию без свободных переменных некоторым статически известным адресом, одинаковым для всех инстансов приложения. Далее можно сконструировать сериализуемое замыкание путём последовательности частичных применений:

```haskell
data Closure a where
  StaticPtr :: StaticPtr b -> Closure b
  Encoded :: ByteString -> Closure ByteString
  Ap :: Closure (b -> c) -> Closure b -> Closure c

main = send "some-node" $
  closure (static factorial) `closureAp` closurePure 10
```

Подробнее можно прочитать в основополагающей статье про облачный Haskell Epstein et al. [2011]. С практической точки зрения — в книжке [Marlow, 2011, глава 16].

Упражнение 43 *Нужно ли явно добавлять в замыкание свободную переменную (∗) (оператор умножения) в реализации факториала?*

5.4 Tagless final интерпретаторы

Как мы убедились ранее (5.1.2), программирование состоит из написания новых и новых интерпретаторов поверх друг друга. Интерпретаторы задают семантику новых языков (5.1.3). В классическом виде язык задаётся как множество деревьев, а интерпретатор отправляет деревья в объект мета-языка. Если язык встроенный, то такой подход называют deep embedding (см. 5.1.4), а соответствующий интерпретатор инициальным.

$$[\![\bullet]\!] : L \to D$$

Можно заметить, что в конечном итоге мы используем только элемент домена, в который интерпретатор отображает программу. Сама программа же представляет собой лишь удобную синтаксическую запись элемента домена и является промежуточным шагом, а не самоцелью. В то же время доменом в случае встроенных языков, заданных интерпретаторами, являются объекты мета-языка. Можем ли мы миновать стадию интерпретации собственного синтаксиса и сразу строить объект домена в синтаксисе мета-языка? Да, такое встраивание называется shallow embedding (5.1.4), о нём эта глава.

5.4.1 Разные интерпретации для shallow embedding

Как мы узнали ранее (4.4), любую структуру данных можно представить свёрткой. И, более того, в итоге можно обойтись без единого конструктора данных (как в списке Чёрча, например): алгебра представляется набором функций, каждая из которых отвечает за сворачивание определённого конструктора.

[126] https://blog.ocharles.org.uk/blog/guest-posts/2014-12-23-static-pointers.html
[127] https://hackage.haskell.org/package/distributed-closure

```
Fix f ≅ forall a . (f a -> a) -> a
List e ≅ forall a . (e -> a -> a) -> a -> a
```

Таким образом, вместо того, чтобы конструировать дерево языка, а затем его интерпретировать (сворачивать), мы можем сразу сконструировать терм типа `∀a . (f a -> a) -> a`. Предоставив ему тип домена `a` и алгебру `f a -> a` (либо в виде пачки функций), мы немедленно получим элемент нужного домена. Чтобы задать другую интерпретацию, нужно передать другой тип домена и алгебру:

```
example :: (Int -> a) -> (a -> a -> a) -> a
example cnst plus = cnst 1 `plus` cnst 41

ghci> example show (\l r -> l ++ " + " ++ r)
"1 + 41"
```

Если зафиксировать интерпретацию, то функции-аргументы можно реализовать статически и просто ссылаться на них в терме. Таким образом, про объявление функций можно думать как про расширение некоторого встроенного предметного языка. Общие рассуждения про shallow embeddings, свёртки и библиотеки можно почитать в Gibbons [2013], Gibbons and Wu [2014]. Сравните:

Deep	Shallow
Синтаксис языка задаётся набором допустимых нод дерева	Декларация функции задаёт новую ноду дерева: вызов этой функции
Интерпретатор при виде каждой ноды выполняет соответствующий код на метаязыке (ветку паттерн-матчинга) после вычисления поддеревьев	Интерпретатор при виде вызова выполняет код тела функции после вычисления аргументов

5.4.2 Дойти до конца

Вернёмся к некаррированной версии свёрток: теперь это тип, принимающий кортеж функций. А кортеж функций можно заменить на класс типов. Тогда декларация класса будет задавать синтаксис встроенного языка, а инстансы для доменов — реализацию. Этот подход называется **tagless final encoding**[128] и фактически это кодирование данных по Чёрчу с классами типов и набором трюков Carette et al. [2007], Kiselyov [2012a].

Снова рассмотрим язык со сложением:

```
data Expr = Const Int | Plus Expr Expr

eval :: Expr -> Int
eval = \case Const x -> x; Plus l r -> eval l + eval r
```

[128] https://okmij.org/ftp/tagless-final/

Или через катаморфизм:

```
data ExprF rec = Const Int | Plus rec rec

eval :: Fix ExprF -> Int
eval = cata \case Const x -> x; Plus l r -> l + r
```

Соответствующее tagless final кодирование будет выглядеть следующим образом:

```
class Expr domain where
  cnst :: Int -> domain
  plus :: domain -> domain -> domain

instance Expr Int where
  cnst x = x
  plus l r = l + r
```

Теперь мы можем сконструировать обычный терм Haskell, задать домен, и машинерия классов типов подставит нужную алгебру самостоятельно:

```
example :: forall domain . Expr domain => domain
example = cnst 1 `plus` cnst 41

ghci> example :: Int
42
```

Чтобы добавить ещё интерпретацию, реализуем инстанс для другого домена:

```
instance Expr String where
  cnst x = show x
  plus l r = l <> " + " <> r

ghci> example :: String
"1 + 41"
```

5.4.3 Восстановление композиционности семантики

Семантика называется **композиционной**, если семантика конструкций зависит только от семантик подконструкций (см. 5.1.3). Иначе говоря, она может быть задана катаморфизмом и, соответственно, инстансом класса типов в tagless final. То есть, чтобы уметь для любой семантики построить tagless final реализацию, нужно уметь универсальным образом превращать некомпозиционные семантики в композиционные.

Всякие преобразования кода, как правило, не композиционные. Для примера рассмотрим протаскивание унарных отрицаний:

```haskell
data Expr1 = Lit Int | Add Expr1 Expr1 | Neg Expr1

transform1 :: Expr1 -> Expr1
transform1 = \case
  Lit x -> Lit x
  Add l r -> Add (transform1 l) (transform1 r)
  Neg (Lit x) -> Lit (-x)                                          -- проблема
  Neg (Neg e) -> transform1 e                                      -- проблема
  Neg (Add l r) -> Add (transform1 (Neg l)) (transform1 (Neg r)) -- проблема
```

Чтобы восстановить композиционность семантики, нужно экплицировать контекстные зависимости с помощью стрелочного домена Kiselyov [2012a]:

```haskell
data Ctx = CtxPos | CtxNeg
flipCtx = \case CtxPos -> CtxNeg; CtxNeg -> CtxPos

transform1' :: Expr1 -> (Ctx -> Expr1)
transform1' expr = case expr of
  Lit x -> \case CtxNeg -> List (-x); CtxPos -> Lit x
  Neg e -> \ctx -> transform1' e (flipCtx ctx)
  Add l r -> \ctx -> Add (transform1' l ctx) (transform1' r ctx)
```

Отсюда можно получить tagless final версию:

```haskell
class Expr2 d where
  lit :: Int -> d
  add :: d -> d -> d
  neg :: d -> d

instance Expr2 d => Expr2 (Ctx -> d) where
  lit x = \case CtxNeg -> lit (-x); CtxPos -> lit x
  neg e = \ctx -> neg e (flipCtx ctx)
  add l r ctx = add (l ctx) (r ctx)
```

5.4.4 Typed tagless final interpreter

Рассмотрим наш пример tagless initial encoding 5.2.3:

```haskell
data Expr ty where
  Const :: Int -> Expr Int
  IsZero :: Expr Int -> Expr Bool
  If :: forall ty . Expr Bool -> Expr ty -> Expr ty -> Expr ty

eval :: Expr ty -> ty
```

```haskell
eval = \case
  Const x  -> x
  IsZero t -> eval == 0
  If c t e -> if eval c then eval t else eval e
```

Чтобы получить final encoding, параметризуем домен результирующим типом выражения:

```haskell
class Expr (domain :: Type -> Type) where
  cnst :: Int -> domain Int
  isZero :: domain Int -> domain Bool
  if' :: forall ty . domain Bool -> domain ty -> domain ty -> domain ty

instance Expr Identity where
  cnst x = Identity x
  isZero (Identity x) = Identity (x == 0)
  if' (Identity c) t e = if c then t else e
```

Упражнение 44 *Какой домен подойдёт для печати выражения?*

5.4.5 Встречаем старых друзей: `Applicative`, `Monad`

Рассмотрим следующий язык в initial encoding с higher order abstract syntax (см. 5.3.6, 5.3.7). Справа перепишем в tagless final, выбирая подходящие имена для кусков синтаксиса.

```haskell
data Expr s ty where
  Val   :: ty -> Expr s ty
  App   :: Expr s (arg -> res)
        -> Expr s arg
        -> Expr s res

  LetIn :: Expr s ty
        -> (ty -> Expr s ty')
        -> Expr s ty'

  Get   :: Expr s s
  Put   :: s -> Expr s ()
```

```haskell
class Applicative domain where
  pure  :: ty -> domain ty
  (<*>) :: domain (arg -> res)
        -> domain arg
        -> domain res

class Monad domain where
  (>>=) :: domain ty
        -> (ty -> domain ty')
        -> domain ty'

class MonadState s domain where
  get :: domain s
  put :: s -> domain ()
```

Реализуем функцию, модифицирующую значение в нашем новом языке:

```haskell
--
modify :: (s -> s) -> Expr s s
modify f =
  Get `LetIn` \x ->
  Put (f x) `LetIn` \() ->
  Val x
```

```haskell
--                    s -> (s, s)
modify :: (s -> s) -> State s s
modify f =
  get >>= \x ->
  put (f x) >>= \() ->
  pure x
```

И до боли знакомую интерпретацию:

```haskell
eval :: Expr s ty -> s -> (s, ty)
eval = \case


  Val x -> \s -> (s, x)
  App fs xs -> \s1 ->
    let (s2, f) = eval fs s1 in
    let (s3, x) = eval xs s2 in
    (s3, f x)


  LetIn comp k -> \s ->
    let (s', x) = eval comp s in
    eval (k x) s'


  Get -> \s -> (s, s)
  Put s -> \_ -> (s, ())
```

```haskell
newtype State s a = State
  { runState :: s -> (s, a) }

instance Applicative (State s) where
  pure x = State \s -> (s, x)
  fs <*> xs = State \s1 ->
    let (s2, f) = runState fs s1 in
    let (s3, x) = runState xs s2 in
    (s3, f x)

instance Monad (State s) where
  comp >>= k = State \s ->
    let (s', x) = runState comp s in
    runState (k x) s'

instance MonadState s (State s) where
  get = State \s -> (s, s)
  put s' = State \s -> (s', ())
```

Таким образом, *аппликативные функторы — meta-circular язык с аппликацией*, а *монадический bind — это фактически let-in в higher-order синтаксисе*.

Если вспомнить, что let $x = M$ in $N \equiv (\lambda x.\, N)\ M \equiv [x \mapsto M]\, N$, то мы поймём, что монадическое связывание — это подстановка `a -> m b` вычисления `m b` вместо результатов типа `a` в вычислении `m a`. Либо, если в качестве `m` взять дерево выражения, то связывание будет подстановкой поддеревьев вместо переменных:

```haskell
data Expr var = Var var | Empty | Append (Expr var) (Expr var)
instance Monad Expr where
  (>>=) :: Expr var -> (var -> Expr var') -> Expr var'
  Var name >>= subst = subst name
  Empty >>= _ = Empty
  Append l r >>= subst = Append (l >>= subst) (r >>= subst)
```

Изначально использовать теор-категорное понятие монады[129] в λ-исчислении было предложено в Moggi [1988], чтобы удобнее записывать денотационную семантику[130]. Это оказалось настолько удобно в моменте, что их стали использовать повсеместно в функциональном программировании, чтобы расширять простые функциональные языки различными могущественными возможностями без изменения самих языков Wadler [1990, 1992]. Был сформулирован Moggi's principle:

«Computations of type α correspond to values of type $f\ \alpha$»

Использовать аппликативные функторы предложили существенно позже, чтобы избежать именования промежуточных шагов вычислений, когда в этом нет необходимости McBride and Paterson [2008]. Таким образом, аппликативы дают встроенный язык выражений, а монады — язык стейтментов.

5.5 Expression problem

Expression problem[131] или **проблема выразительности** — это некоторый критерий выразительности языка программирования, сформулированный Wadler'ом в 1998[132]. Ставится вопрос: насколько легко расширять синтаксис встроенного языка и добавлять новые интерпретации? Иначе говоря, насколько легко добавлять новые разновидности данных и методы обработки.

Под "легкостью" подразумевается локальность: нужно ли править различные куски кода для этого. Например, если синтаксис языка задан обычным алгебраическим типом данных, то добавить новую интерпретацию "легко" — просто добавить новую рекурсивную функцию, а добавить новую синтаксическую конструкцию — "сложно" — нужно изменить все интерпретаторы:

```haskell
data Expr
  = Const Int
  | Plus Expr Expr -- добавляем

eval :: Expr -> Int
eval = \case Const x -> x; Plus l r -> eval l + eval r

show :: Expr -> String
show = \case Const x -> show x; Plus l r -> show l ++ " + " ++ show r
```

Если язык задан, например, с помощью наследования, то, наоборот, расширить синтаксис легко — добавить новый класс, а добавить интерпретацию сложно — добавить реализацию метода в каждом классе:

[129] https://ncatlab.org/nlab/show/monad+(in+computer+science)
[130] Часто сопутствующее монадам понятие эффекта мы рассмотрим далее 7.
[131] https://en.wikipedia.org/wiki/Expression_problem
[132] https://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt

```
interface Lang {
    fun eval(): Int
    fun show(): String
}
class Const(val x: Int) : Lang {
    override fun eval() = x
    override fun show() = x.toString()
}
class Plus(val l: Lang, val r: Lang) : Lang {
    override fun eval() = l.eval() + r.eval()
    override fun show() = "$l + $r"
}
```

Оказывается, существуют подходы, позволяющие добиться "лёгкости" по обоим измерениям. Мы уделим им много внимания в этом курсе.

Действительно, как мы обсуждали ранее 5.1.1, программы представляют собой серию интерпретаторов. Для всё той же борьбы со сложностью, важно уметь описывать эти интерпретаторы модульно — задавать части языков отдельно друг от друга, и собирать нужные языки по месту из готовых блоков. Это помогает составлять программы из простых переиспользуемых компонент, каждая из которых имеет чёткую зону ответственности.

Expression problem возникала и решалась много раз: expression problem, stable denotations, extensible (modular) interpreters. Прошло немало времени, пока не возникло понимание, что всё это об одном и том же[133].

5.5.1 Копроизведение функторов

Воспользуемся представлением данных как неподвижной точки функтора (см. 4.3.2). В качестве модельного языка возьмём язык выражений со сложением:

```
data Basic rec = Const Int | Plus rec rec

algBasic :: Basic Int -> Int
algBasic = \case Const x -> c; Plus l r -> l + r

evalBasic :: Fix Basic -> Int
evalBasic = cata algBasic
```

Заметим, что сумма (копроизведение) функторов формы даёт функтор формы, алгебра для которого получается из алгебр компонент[134]:

```
data (l :+: r) rec = L (l rec) | R (r rec)
```

[133] https://okmij.org/ftp/Computation/having-effect.html

[134] Пользуемся расширением TypeOperators.

```haskell
(\/) :: (l a -> a) -> (r a -> a) -> ((l :+: r) a -> a)
phi \/ psi = \case L l -> phi l; R r -> psi r
```

Расширим наш язык чтением числа из окружения. Следуя рассмотренной ранее денотационной семантике 5.1.3, выберем функцию `Int -> Int` в качестве домена:

```haskell
data Basic rec = Const Int | Plus rec rec
data Input rec = Input

algBasic' :: Basic (Int -> Int) -> Int -> Int
algBasic' = \case Const x -> \_env -> x; Plus l r -> \_env -> l + r

algInput :: Input (Int -> Int) -> Int -> Int
algInput = \case Input -> \env -> env
```

Таким образом, мы добились возможности отдельно определять куски синтаксиса и семантики языка, и собирать нужный язык по месту. Напишем программу на нашем языке с неявным иммутабельным состоянием:

```haskell
f :: Int -> Int
f = cata (algBasic' \/ algInput) $
  In (L (Plus (In (L (Const 1))) (In (R Input)))) -- 1 + input
```

Однако заметим, что пока мы не решили проблему полностью, так как интерпретация новой конструкции `Input` потребовала более сложный домен и нам пришлось переписывать интерпретацию старой для него: из `algBasic` в `algBasic'`. Теперь мы понимаем, почему stable denotations — это ещё одно название для expression problem 5.5. Далее мы дополним это решение до полноценного 7.

5.5.2 Произведение алгебр

Построим tagless final представление для интерпретатора из предыдущего параграфа. Теперь синтаксис задаётся классами типов:

```haskell
class Basic domain where
  cnst :: Int -> domain
  plus :: domain -> domain -> domain

class Input domain where
  input :: domain
```

В качестве домена, как и ранее возьмём функции `Int -> Int` и реализуем интерпретацию:

```haskell
instance Basic (Int -> Int) where
  cnst value = \_ -> value
  plus l r = \env -> l env + r env

instance Input (Int -> Int) where
  input = \env -> env
```

Теперь, чтобы собрать язык по месту, можно просто воспользоваться композируемостью контекстов классов типов, всю остальную работу:

```haskell
example :: forall domain . (Basic domain, Input domain) => domain
example = cnst 1 `plus` input

ghci> (example :: Int -> Int) 41
42
```

6 Продолжения (continuations)

Продолжения с начала 60х не один раз возникали в литературе в различных формах и разнообразных приложениях Reynolds [1993], Landin [1997], пока в 70х Wadsworth не придумал общий термин и единую концепцию — **continuation**[135] — "the meaning of the rest of the program".

Начальным толчком к размышлениям стал язык Algol 60, имевший нетривиальный механизм меток и прыжков. Проблемой была как имплементация семантики, так и её денотационное описание вместе с трансляцией в лямбда-исчисление. Действительно, как математически описать `goto`? В каком домене искать семантику таких программ? Как написать определяющий интерпретатор, отправляющий программу в этот домен? Решением стала возможность сослаться на семантику остатка программы, продолжение, в определённой точке (например, на метке).

О продолжениях всегда должен думать автор языка, ведь вычислителю в каждой точке программы нужно знать, что исполнять дальше. А как мы уже поняли, любой программист является автором множества языков, приближающих его к решению задачи. Более того, многие повседневные языковые конструкции явно или неявно непосредственно манипулируют продолжениями.

6.1 Концепция продолжений

Рассмотрим семантику выражения с операционной точки зрения, как последовательность шагов переписывания (см. 5.1.3). Если внимательно рассмотреть каждый шаг, то мы обнаружим, что он состоит из двух этапов: поиска подвыражения (редекса), в котором можно сделать элементарный шаг вычислений, выполнение этого шага, и так далее[136]. Во время поиска редекса, выражение разбивается на две части (см. 18):

- Фокус — подвыражение в котором ищем редекс;
- Продолжение — остаток выражения с "дыркой", обозначающий место, куда нужно подставить результат шага вычислений.

Как правило, продолжения существуют вне пользовательского кода как состояние интерпретатора, которому в каждый следующий момент времени нужно помнить, какой код и в каком состоянии исполнять дальше. Однако, языки предоставляют пользователям множество конструкций, позволяющих управлять продолжениями (см. 19):

- Функция `exit` выбрасывает продолжение программы целиком;
- Конструкция `try-catch` позволяет выбросить часть продолжения до места поимки исключения и исполнить оставшееся;

[135] `https://en.wikipedia.org/wiki/Continuation`

[136] На самом деле вычислениям с продолжениями учат в начальных классах, когда рассказывают про вычисление выражений "по действиям".

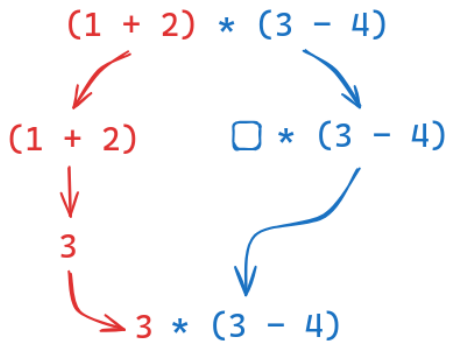


Рис. 18: Выражение разделяется на фокус (красный) и продолжение (синее), когда в фокусе удалось сделать шаг, они объединяются в новое выражение, для которого процесс повторяется.

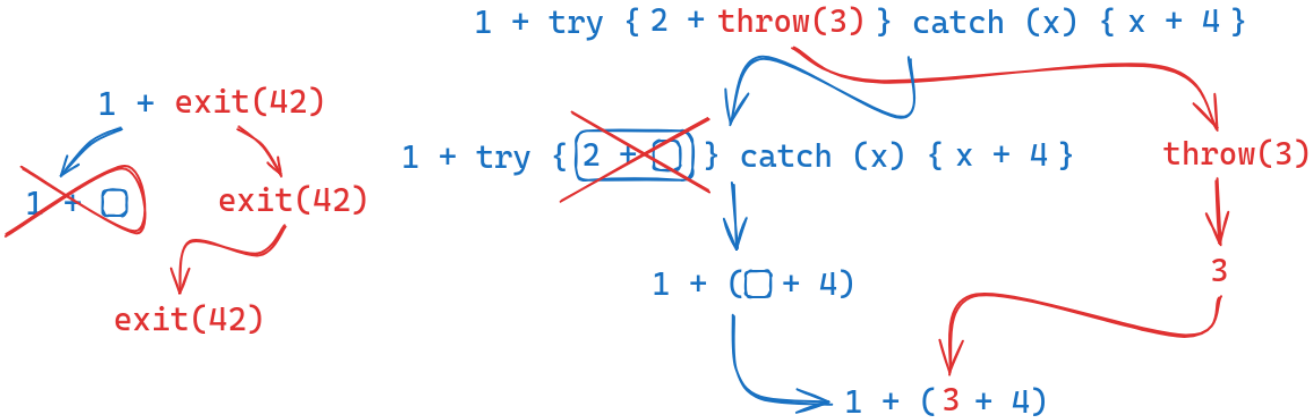


Рис. 19: Конструкция `exit` выбрасывает продолжение и программа останавливается, в то время как `throw` выбрасывает лишь часть продолжения до ближайшего `try-catch`.

- Конструкция `return` позволяет восстановить исполнение в месте, где функция была вызвана;
- Конструкции `break` и `continue` восстанавливают продолжение после цикла и до...

6.1.1 Reduction semantics

Стиль задания операционной семантики через описание поиска примитивных ("**головных**") редексов и их редукций называют **семантикой редукционных контекстов (reduction semantics)**[137]. Как обычно, следуя Hutton's Razor, рассмотрим интерпретацию маленького простого язычка с вычитанием (несимметричная операция для проверки реализации):

```
1 data Expr = Const Int | Diff Expr Expr
```

[137] https://en.wikipedia.org/wiki/Operational_semantics

Для начала нам нужно определить синтаксис продолжений, “выражений с дыркой”. То есть с технической точки зрения продолжение — это структура данных, содержащая всю необходимую информацию, чтобы продолжить исполнение (здесь — только остаток выражения):

```
data K
  = Hole          -- дырка □
  | LDiff K Expr -- фокус пошел в правое подвыражение, запомнили левое
  | RDiff Int K  -- посчитали левое, фокус пошел вправо
```

Зададим операцию разбиения выражения на продолжение и фокус:

```
split :: Expr -> (Expr, K)
split e = case e of
  Const _ -> (e, Hole)
  Diff (Const _) (Const _) -> (e, Hole) -- примитивный редекс
  Diff (Const l) r ->                   -- левое подвыражение уже посчитано
    let (focus, k) = split r in         -- ищем редекс в правом
    (focus, RDiff l k)
  Diff l r ->                           -- ничего ещё не посчитано
    let (focus, k) = split l in         -- начинам с поиска редекса слева
    (focus, LDiff k r)

ghci> split (Diff (Diff (Const 1) (Const 2)) (Const 3)) -- (1 - 2) - 3
(Diff (Const 1) (Const 2), LDiff Hole (Const 3)) -- (1 - 2, □ - 3)
```

Шаг примитивной редукции умеет только вычитать числа:

```
headReduction :: Expr -> Expr
headReduction = \case
  Diff (Const l) (Const r) -> Const (l - r)
  e -> e
```

После шага примитивной редукции нам понадобится подставить результат обратно в продолжение, чтобы из “выражения с дыркой” получить полноценное выражение, которое можно продолжить редуцировать:

```
plugIn :: Expr -> K -> Expr
plugIn e k = case k of
  Hole -> e
  LDiff k' r -> Diff (e `plugIn` k') r
  RDiff l k' -> Diff (Const l) (e `plugIn` k')
```

Как правило, подстановку терма t в продолжение E обозначают как $E[t]$.

Теперь мы можем определить полноценный шаг:

```haskell
transition :: Expr -> Expr
transition e =
  let (focus, k) = split e in    -- разбиваем на фокус и контекст
  headReduction focus `plugIn` k -- делаем вычисление в фокусе и подставляем
```

Тода операционная семантика это развёртка списка промежуточных выражений:

```haskell
eval :: Expr -> [Expr]
eval = List.unfoldr \prev ->
  let next = transition prev in
  if prev == next then Nothing else Just (next, next)
```

Можно заметить, что `split` и `plugIn` однозначно определяется по синтаксису продолжений и виду главных редукций, а `transition` и `eval` одинаковы для произвольного языка. Поэтому семантику редукционных контекстов обычно задают как синтаксис продолжений, перечень главных редукций и единственное правило вывода — шаг в контексте:[138]

Values	v	$::= \mathbb{Z}$
Terms	t	$::= v \mid t \ominus t$
Evaluation context	E	$::= \square \mid E \ominus t \mid \mathbb{Z} \ominus E$
(diff)		$v_1 \ominus v_2 \longrightarrow v_1 - v_2$
step		$\dfrac{t \longrightarrow t'}{E[t] \longrightarrow E[t']}$

Существует стандартный инструмент PLT Redex[139] для описания и тестирования семантики в стиле редукционных контекстов.

6.1.2 Continuation semantics

Запишем денотационную семантику нашего языка:

```haskell
evalDirect :: Expr -> Int
evalDirect = \case
  Const n -> n
  Diff l r -> evalDirect l - evalDirect r
```

Перепишем денотационную семантику в стиле с явными продолжениями. Но сначала поработаем с типом `K` и упростим работу с ним:

```haskell
data K = Hole | LDiff K Expr | RDiff Int K
-- K = 1 + (K × Expr) + (Int × K) = 1 + (Expr + Int) × K
data Frame = LDiff Expr | RDiff Int
type K = [Frame]
```

[138] $\ominus$ тут обозначает синтаксический минус (ноду в дереве).
[139] `https://redex.racket-lang.org/`

Получили представление продолжения как стека фреймов.

Денотационную семантику будем записывать для домена `K -> Int`. Такая разновидность денотационной семантики с явным представлением смысла остатка программы (в виде `K`) иногда называют **continuation semantics**. А соответствующий стиль программирования с передачей продолжений — **continuation passing style (CPS)**.

В нашем continuation semantics будет выглядеть как пара взаимно-рекурсивных функций:

```
evalK :: Expr -> K -> Int
evalK e k = case e of
  Const n -> k `appK` n              -- выполняем остаток программы
  Diff l r -> evalK l (LDiff r : k) -- запоминаем дальше вычислить правое

appK :: K -> Int -> Int
appK k result = case k of
  [] -> result                                  -- дальше делать нечего
  LDiff r : k' -> evalK r (RDiff result : k') -- идем вычислять вправо
  RDiff l' : k' -> k' `appK` (l' - result)    -- продолжаем на результате
```

Тут мы снова спускаемся по выражению в поисках примитивного редекса (тут — константы), попутно запоминая, что нужно будет сделать после.

Первая рекурсивная реализация `evalDirect` не заботилась о продолжениях. Однако, продолжения — это неотъемлемая часть процесса вычисления, вычислителю нужно знать, что делать дальше в каждый момент. На самом деле `evalDirect` лишь делегирует работу с продолжениями определяемого языка мета-языку, но каким образом? Заметьте, что полученная в итоге реализация `evalK` хвостово-рекурсивная, а значит, может быть скомпилирована в цикл, не потребляющий стек вызовов, при этом `K` это стек. Таким образом, в первом случае мы делали рекурсивные вызовы и продолжение аллоцировалось на (аппаратном) стеке мета-языка, а во втором случае мы самостоятельно аллоцируем стек в куче.

В случае, если стек мета-языка реализован поверх аппаратного, есть риск ошибки его переполнения. Чтобы этого избежать, используется техника **trampolining**, которая как раз состоит в ручной аллокации продолжения в куче Ganz et al. [1999], Bjarnarson [2012].

Подобно тому как тип контекста зиппера можно вычислить как производную алгебраического представления соответствующего типа Huet [1997], McBride [2001], Abbott et al. [2003], так можно вычислить тип продолжения свёртки McBride [2008].

6.1.3 Продолжения первого класса

В примерах выше конструкции языка управляют продолжениями неявно (см. рис. 19). Однако, иногда вводят операторы, позволяющие явно оперировать продолжениями. С их помощью можно реализовать как возможности манипуляции потоком управления вроде генераторов и корутин[140][141], так и все остальные эффекты вроде состояния (см. далее 7).

[140] https://en.wikibooks.org/wiki/Haskell/Continuation_passing_style#Example:_coroutines
[141] https://kotlinlang.org/api/core/kotlin-stdlib/kotlin.coroutines/suspend-coroutine.html

Продолжения первого класса (first-class continuations) — продолжения, которые представимы в программе в виде значений. Учитывая, что продолжение имеет вакантное место ещё не вычисленного подвыражения, продолжения первого класса представляют функциями первого класса.

Чтобы получить в коде продолжение первого класса, нужно либо написать код в CPS, либо воспользоваться встроенным в язык оператором, которых придумано великое множество [Hillerström, 2022, приложение A]. Например, J, `escape` Reynolds [1972], `call/cc`...

Для примера реализуем в языке операцию `Cont`, позволяющую захватить текущее продолжение. Он будет принимать пользовательскую функцию и передавать в неё текущее продолжение от себя и до конца программы:

$$E[cont\ f] \longrightarrow f\ (\lambda x.\, E[x])$$

Сначала расширим язык лямбда-исчислением:

```haskell
data Expr
  = Const Int | Diff Expr Expr
  | Var String | Lam String Expr | App Expr Expr

data Frame
  = LDiff Expr | RDiff Value
  | LApp Expr | RApp Value
  | SetEnv Env -- после исполнения замыкания возвращаем текущее окружение
```

Реализация интерпретатора довольно прямолинейна. Единственное, сейчас остаток программы представлен в виде линейного списка, который фактически обходится в цикле последовательно. Нужно не забыть после исполнения тела замыкания восстановить изначальное окружение для исполнения продолжения:[142]

```haskell
evalK :: (?env :: Env) => Expr -> K -> Value
evalK e k = case e of
  Const n -> k `appK` Number n
  Diff l r -> evalK l (LDiff r : k)
  Var name -> k `appK` (?env ! name)
  Lam name body -> k `appK` Closure name ?env body
  App f arg -> evalK f (LApp arg : k)

appK :: (?env :: Env) => K -> Value -> Value
appK k result = case k of
  [] -> result
  LDiff r : k' -> evalK r (RDiff result : k')
  RDiff l' : k' -> k' `appK` Number (unwrapNumber l' - unwrapNumber result)
```

[142] Для удобства буддем передавать окружение неявным параметром.

```
  LApp arg : k' -> evalK arg (RApp result : k')
  RApp f : k' -> case f of
    Closure name env body ->
      let currEnv = ?env in
      let ?env = Map.insert name result env in
      evalK body (SetEnv currEnv : k')
    K env k'' ->
      let currEnv = ?env in
      let ?env = env in
      (k'' ++ SetEnv currEnv : k') `appK` result
    other -> error $ "Expected callable, got " <> show other
  SetEnv env : k' -> let ?env = env in k' `appK` result
  where
    unwrapNumber = \case
      Number n -> n
      other -> error $ "Expected number, got " <> show other
```

Наконец, реализуем оператор cont:

```
data Expr = ... | Cont Expr
data Frame = ... | ContFrame

evalK e k = case e of
   ...
   Cont f -> evalK f (ContFrame : k)

appK k result = case k of
   ...
   ContFrame : k' -> [RApp result] `appK` K ?env k' -- переиспользуем ветку RApp
```

Рассмотрим несколько примеров (обозначим `contLam name body = Cont (Lam name body)`, `(-.) = Diff`, `c = Const`, `v = Var` и `(@) = App`):

- `c 10 -. contLam "k" (c 1)` — выбросить продолжение и вернуть 1 ($k = 10 - \square$);
- `c 10 -. contLam "k" (v "k" @ c 1)` — $\rightsquigarrow k\ 1 \rightsquigarrow 10 - 1 \rightsquigarrow 9$;
- `c 10 -. contLam "k" (v "k" @ c 1 -. c 2)` — $\rightsquigarrow k\ 1 - 2 \rightsquigarrow 9 - 2 \rightsquigarrow 7$;
- `c 10 -. contLam "k" (v "k" @ c 1 -. v "k" @ c 2)` — $\rightsquigarrow k\ 1 - k\ 2 \rightsquigarrow 9 - 8 \rightsquigarrow 1$.

Таким образом, мы получили язык, который позволяет пользователю в произвольном месте программы получить текущее продолжение в виде функции.

Возможность получить продолжение первого класса почти не предоставляется промышленными языками, так как это довольно опасный инструмент. Действительно, если пользователь не вызовет продолжение, может не произойти закрытия ресурсов. Если вызовет несколько раз, снова может произойти как некорректная работа с ресурсами, так и порча

изменяемой памяти (продолжения, которые можно безопасно вызывать много раз называют **multi-shot**). Подобные поведения можно исключать специальной обработкой таких ситуаций Muhcu et al. [2025], проверками времени компиляции (например, с помощью линейных типов) или времени исполнения. Сейчас с активными исследованиями о внедрении хендлеров эффектов (см. далее 7) продолжения первого класса могут получить новый шанс.

6.2 Продолжения своими руками

Рассмотрим, как выглядит CPS в обычном коде, а не в контексте deep embedding. Для этого мы, во-первых, перейдём к shallow embedding и будем сразу строить элементы целевого домена. А во-вторых, рефункционализируем продолжения `K` и обобщим: вместо `K -> Int` будем использовать `forall r . (a -> r) -> r`, где `a -> r` — функциональное представление продолжения.

Такой CPS эксплуатирует следующий изоморфизм:

```
to :: a -> (forall r . (a -> r) -> r)
to x k = k x

from :: (forall r . (a -> r) -> r) -> a
from comp = comp id
```

Иначе говоря, вместо того, чтобы предоставить значение типа `a`, можно спросить у вызывающей стороны, как она собирается с этим значением работать `a -> r`, сделать это самостоятельно, и вернуть вызывающей стороне `r`.

Прикладным программистам этот изоморфизм знаком по технике использования callback'ов. Теоретикам же известно, что он является частным случаем леммы Йонеды Hinze and James [2010]. Сравните CPS с интансом этой леммы в теории предпорядка (выше мы считали `a = Unit; b = a`):

$$a \to b \cong \forall r.\, (b \to r) \to (a \to r)$$
$$a \leq b \iff \forall r.\, (b \leq r) \Rightarrow (a \leq r)$$

Например, мы можем переписать факториал в CPS. Заметьте, что код имеет доступ к продолжению первого класса (однако, пока никак нетривиально не использует его), при этом снова является хвостово-рекурсивным.

```
facCps :: Int -> (forall r . (Int -> r) -> r)
facCps n k
  | n <= 1 = k 1
  | otherwise = facCps (n - 1) \res -> k (n * res)

facCps 3 id ⇝ facCps 2 \res -> id (3 * res)
⇝ facCps 1 \res -> id (3 * (2 * res)) ⇝ id (3 * (2 * 1)) ⇝ 6
```

Упражнение 45 *Сколько функция `facCps` потребляет стековой памяти?*

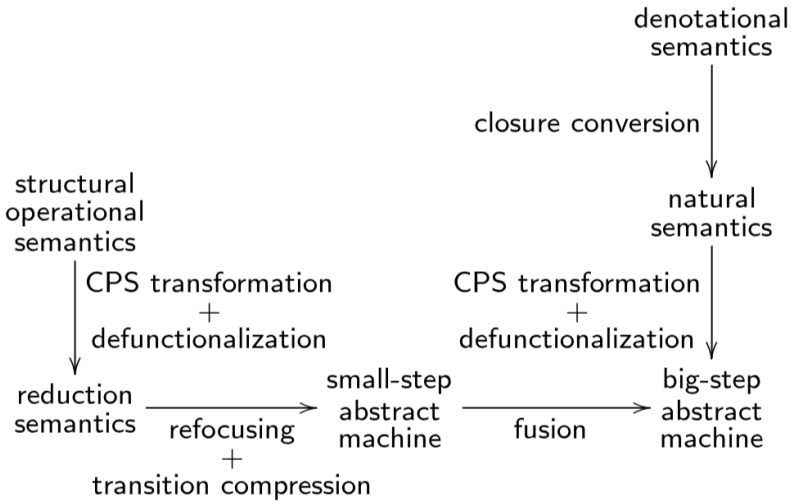


Рис. 20: Связь между семантиками в различных стилях Danvy [2008].

6.2.1 Дефункционализация и аккумуляторы

Дефункционализируем продолжения в `facCps`. Мы используем две функции высших порядков, `id` и `\res -> k (n * res)`, они дают нам два конструктора:

```
data K = Id | Times Int K
-- ≅
type K = [Int]

runK :: K -> Int
runK = product

facCps :: Int -> K -> Int
facCps n k
  | n <= 1 = runK k
  | otherwise = facCps (n - 1) (snoc k n)

facCps 3 [] ⇝ facCps 2 [3] ⇝ facCps 1 [3, 2] ⇝ runK [3, 2] ⇝ 3 * (2 * 1)
```

Мы снова получили представление продолжения в виде стека фреймов. А значит, интерпретатор `evalK` полученный нами ранее (6.1.2) является CPS версией обычного интерпретатора, только с дефункционализированными продолжениями (они удобны для отладки — их можно распечатать, в отличие от функций Haskell). Если расширить тот наш язык и написать на нём факториал, а потом сделать fusion или рефункционализацию (4.4.1) дерева программы и дерева продолжения, мы получим текущую реализацию факториала. И в целом между различными стилями реализаций семантик можно построить соответствие (см. рис. 20).

Этой техникой можно пользоваться и в более сложных случаях, чтобы легко получать

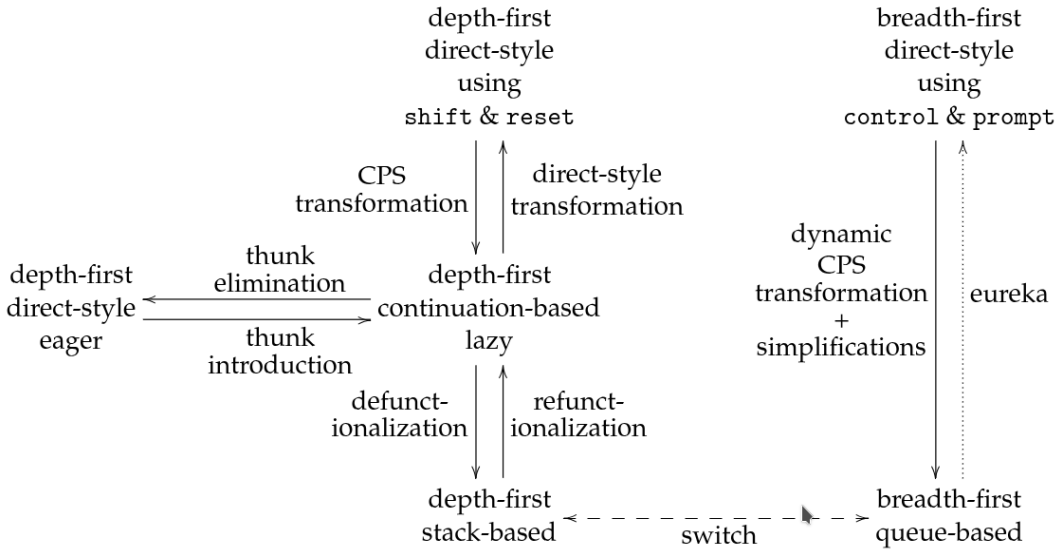


Рис. 21: Трансформации кода, основанные на CPS и де(ре)функционализации Danvy [2006].

хвостово-рекурсивные (итеративные) реализации[143] Gibbons [2021][144]. В целом CPS и дефункционализация — это богатый источник различных рефакторингов (см. рис. 21).

Теперь заметим, что операция умножения ассоциативна, а значит, можно `snoc` заменить на умножение, а продолжение представить одним числом. Получим привычную реализацию факториала с аккумулятором:

```
facAcc :: Int -> Int -> Int
facAcc n acc
  | n <= 1 = acc
  | otherwise = facAcc (n - 1) (acc * n)

facAcc 3 1 ⇝ facAcc 2 (1 * 3) ⇝ facAcc 1 ((1 * 3) * 2) ⇝ (1 * 3) * 2 ⇝ 6
```

Также, можно фреймы представить в виде эндоморфизмов и в качестве ассоциативной операции использовать композицию функций Ploeg and Kiselyov [2014][145].

6.2.2 Monad Cont

Из-за CPS код потерял привычную структуру, при которой функции напрямую возвращают свои результаты (i.e. **direct style**). При наличии большого количества вызовов трансформированных функций, код становится плохо читаемым (проблема известна как **callback hell**):

```
fibCps :: Int -> (forall r . (Int -> r) -> r)
```

[143] https://www.pathsensitive.com/2019/07/the-best-refactoring-youve-never-heard.html
[144] (youtube) Jeremy Gibbons - Continuation-passing style, defunctionalization, and associativity.
[145] https://wiki.haskell.org/Difference_list

```haskell
fibCps n k = if n <= 2 then k 1 else
  fibCps (n - 1) \res1 ->
  fibCps (n - 2) \res2 ->
  k (res1 + res2)
```

Домен `(a -> r) -> r` можно сделать монадой и восстановить direct style код внутри do-нотации. Заведём `newtype` обёртку для объявления инстансов:

```haskell
newtype Cont r a = Cont { runCont :: (a -> r) -> r }
```

Функтор добавляет пост-процессинг результату перед передачей в продолжение:

```haskell
instance Functor (Cont r) where
  -- fmap :: (a -> b) -> ((a -> r) -> r) -> ((b -> r) -> r)
  fmap f (Cont comp) = Cont \k -> comp (k . f)
```

Аппликатив просто передаёт значение продолжению:

```haskell
instance Applicative (Cont r) where
  pure x = Cont \k -> k x
  (<*>) = ap
```

Можно заметить, что монадическое связывание вторым аргументом тоже принимает продолжение, но “маленькое”, до конца do-блока. Таким образом, смысл реализации монадического связывания для `Cont` — это композиция “маленького” продолжения с “большим” продолжением, передаваемым снаружи:

```haskell
instance Monad (Cont r) where
  (>>=) :: Cont r a -> (a -> Cont r b) -> Cont r b
  Cont comp >>= k = Cont \k' -> comp \x -> runCont (k x) k'
```

Теперь мы можем писать линейный код, а монадическая машинерия сама конструирует продолжения и подкладывает в предыдущие вычисления:

```haskell
fibCont :: Int -> Cont r Int
fibCont n = if n <= 2 then pure 1 else do
  res1 <- fibCont (n - 1)
  res2 <- fibCont (n - 2)
  pure (res1 + res2)
```

Упражнение 46 *Оборвите вычисление, если `res1` больше 50.*

Упражнение 47 *Оборвите вычисление как только общий результат стал больше 50.*

Монада `Cont` даёт реализацию встроенного языка, в котором можно получить продолжение вычисления:

```haskell
Cont :: ((a -> r) -> r) -> Cont r a
```

Прикладным программистам такая техника написания CPS кода через монады знакома в виде концепций `Future/Promise`[146][147].

6.2.3 `call/cc`

Самым известным классическим оператором, который использовали в Scheme для получения продолжений первого класса, является `call/cc` **(call with current continuation)**[148][149]. Предоставляемое продолжение является **неограниченным (undelimited/abortive)**, так как оно содержит "конец программы" — никакой код не будет исполняться после его вызова. Неограниченные продолжения де-факто — не совсем функции, так как они не возвращают результата (он уже "beyond the grave"), следовательно, они также не композируются (также как странно композировать `abort` с `exit`)[150][151]. Они скорее являются ко-значениями: пока часть программы выполняется, они ожидают её результата Curien and Herbelin [2000].

Давайте сэмулируем `call/cc` в монаде `Cont`. Захватить всё продолжение программы у нас не выйдет, так как оно собирается только в рамках `Cont`, но мы можем проигнорировать продолжение вызова захваченного продолжения:

```haskell
callCC :: ((a -> Cont r b) -> Cont r a) -> Cont r a
callCC f = Cont \k -> runCont (f \x -> Cont \_ -> k x) k

foo :: Int -> Cont r String
foo x = callCC \k -> do
  let y = x ^ 2 + 3
  when (y > 20) $ k "over twenty" -- throws next line away
  pure (show $ y - 4)
```

6.3 Delimited continuations

В современной практике, как правило, используют продолжения не до конца программы, а только до определённой точки. Их называют **ограниченными** или **разграниченными продолжениями**, **delimited** или **composable continuations**, **subcontinuations**[152][153].

Конструкции для работы с такими продолжениями парные: вводится оператор, ограничивающий текущее продолжение (мб имеющий метку); а также оператор захвата фрагмента

[146] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise
[147] https://github.com/promises-aplus/promises-spec/issues/94
[148] https://en.wikipedia.org/wiki/Call-with-current-continuation
[149] https://en.wikibooks.org/wiki/Haskell/Continuation_passing_style#callCC
[150] https://okmij.org/ftp/continuations/undelimited.html
[151] https://okmij.org/ftp/continuations/against-callcc.html
[152] https://www.cl.cam.ac.uk/teaching/2324/R277/handout-delimited-continuations.pdf
[153] (youtube) Keynote: Delimited Continuations, Demystified by Alexis King | Lambda Days 2023.

$$
\begin{array}{lrcl}
\text{(Variables)} & x, y, \ldots & & \\
\text{(Expressions)} & e & ::= & x \mid \lambda x.e \mid e\ e \\
& & \mid & \mathit{newPrompt} \mid \mathit{pushPrompt}\ e\ e \\
& & \mid & \mathit{withSubCont}\ e\ e \mid \mathit{pushSubCont}\ e\ e
\end{array}
$$

Рис. 22: Синтаксис λ-исчисления с примитивами для работы с продолжениями.

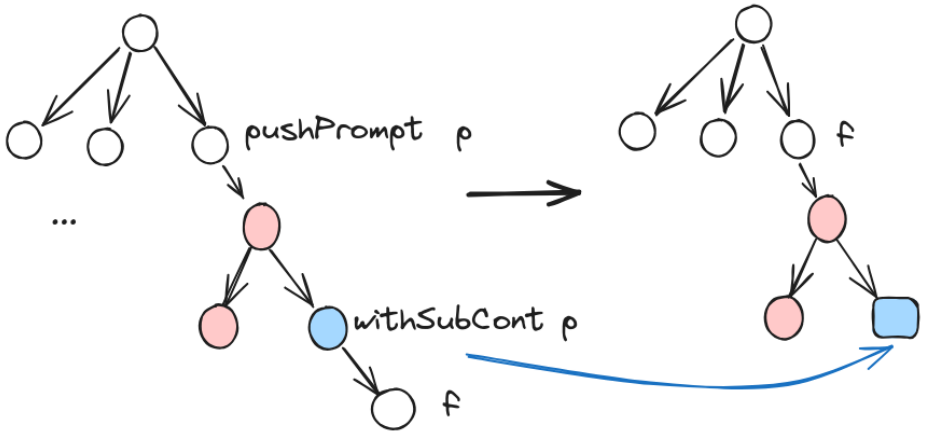


Рис. 23: Пример работы `withSubCont`.

текущего продолжения (мб до конкретного ограничителя с определённой меткой). Таких операторов придумано много [Hillerström, 2022, приложение A], но они более-менее все сводятся друг к другу.

Например, работа с исключениями подразумевает использование двух конструкций: продолжение ограничивается с помощью `try-catch`, а `throw` выкидывает соответствующее частичное продолжение, не захватывая его (см. рис. 19):

$$E_1[try\{E_2[throw(v)]\}catch(x)\{t\}] \to E_1[[x \mapsto v]\ t]$$

Мы же рассмотрим универсальные операторы из Dyvbig et al. [2007]. Работа вводит следующий набор синтаксических конструкций (рис. 22) для работы с ограниченными продолжениями в дополнение к чистому call-by-value лямбда-исчислению:

- $\mathit{newPrompt}$[154] — создаёт свежий идентификатор (метку) ограничения;
- $\mathit{pushPrompt}\ p\ e$ — устанавливает ограничение с меткой p и исполняет выражение e;
- $\mathit{withSubCont}\ p\ f$ — захватывает частичное продолжение до ограничения с меткой p и передаёт в функцию f, возвращает результат f (рис. 23);
- $\mathit{pushSubCont}\ k\ v$ — исполняет композицию текущего продолжения и k на значении v.

[154] Исторически в лиспах неограниченные продолжения были ограничены лишь REPL, отсюда название ограничений — “prompt”.

$$
\begin{array}{l}
(\lambda p.\ 2 + \mathit{pushPrompt}\ p \\
\qquad \text{if } (\mathit{withSubCont}\ p \\
\qquad\quad (\lambda k.\ (\mathit{pushSubCont}\ k\ \text{False})\ +\ (\mathit{pushSubCont}\ k\ \text{True}))) \\
\qquad \text{then } 3 \\
\qquad \text{else } 4) \\
\quad \mathit{newPrompt}
\end{array}
$$

Рис. 24: Пример выражения (результат — 9).

Операторы ограниченных продолжений легко понять как **resumable exceptions**, исключения, которые можно поймать, а программу возобновить с того места, где исключение было выброшено (с некоторым значением). Необычно то, что классические операторы ограниченных продолжений принимают код, работающий с частичным продолжением в месте "кидания исключения", а не в месте "поимки". То есть блок обработки пишется не с `catch`, а с `throw`:

$$
\begin{array}{c}
E_1[\mathit{pushPrompt}\ p\ \{E_2[\mathit{withSubCont}\ p\ f]\}] \to E_1[f\ E_2] \\
E_1[\mathit{try}\{E_2[\mathit{throw}\ v]\}\mathit{catch}(x,k)\{t\}] \to E_1[[x \mapsto v]\,[k \mapsto \lambda y.\, E_2[y]]\ t]
\end{array}
$$

Упражнение 48 *Поредуцируйте пример рис. 24.*

6.3.1 Реализация операторов

Расширим язык 6.1.3 рассмотренными операторами для работы с ограниченными. Для простоты вместо свежих меток промптов будем использовать имена, а вместо `pushSubCont` обычную аппликацию:

```
data Expr = ...| PushPrompt String Expr | WithSubCont String Expr
data Frame = ... | PushPromptFrame String | WithSubContFrame String
```

Поиск редексов расширяется очевидным образом:

```
evalK :: (?env :: Env) => Expr -> K -> Value
evalK e k = case e of
  -- ...
  PushPrompt promptName body -> evalK body (PushPromptFrame promptName : k)
  WithSubCont promptName f -> evalK f (WithSubContFrame promptName : k)
```

Вся задача `PushPrompt` — это застрять в продолжении в виде `PushPromptFrame`, но когда до него доходит исполнение, он просто игнорируется:

```haskell
appK :: (?env :: Env) => K -> Value -> Value
appK k result = case k of
  -- ...
  PushPromptFrame _ : k' -> k' `appK` result
```

Чтобы захватить фрагмент продолжения, нам нужна операция, которая найдёт первый промпт с соответствующим именем в продолжении и вернёт **subcontinuation** — до этого промпта и **metacontinuation** — после:

```haskell
splitByPrompt :: K -> String -> (K, K)
splitByPrompt k targetPromptName = go [] k
  where
    go _ [] =  error $ "Prompt " ++ targetPromptName ++ " not found"
    go subcont (PushPromptFrame promptName : metacont)
      | promptName == targetPromptName = (subcont, metacont)
    go subcont (frame : metacont) = go (subcont ++ [frame]) metacont
```

Теперь `WithSubCont` должен просто применить пользовательскую функцию к subcontinuation, но при этом не забыть дальше исполнить metacontinuation:

```haskell
appK :: (?env :: Env) => K -> Value -> Value
appK k result = case k of
  -- ...
  WithSubContFrame promptName : k' ->
    let (subcont, metacont) = k' `splitByPrompt` promptName in
    (RApp result : metacont) `appK` K ?env subcont
```

Упражнение 49 *Поредуцируйте следующий пример:*

```haskell
exampleDelimited = let ?env = Map.empty in flip evalK [] $
  c 10 -. PushPrompt "p"
    (c 5 -. WithSubCont "p"
      (lam "k" $ v "k" @ c 1 -. v "k" @ c 3))
```

Многие классические операторы, например, `shift/reset`, `prompt/control` и т.д., можно получить, оставив фрейм промпта в subcontinuation или в metacontinuation в реализации `splitByPrompt` Dyvbig et al. [2007].

6.3.2 В `Monad Cont`

Для примера реализуем два классических оператора для работы с ограниченными продолжениями — **shift-reset**. Воспользуемся продолжениями, собираемыми монадой `Cont`. `shift` просто захватывает текущее продолжение и передаёт его в пользовательское вычисление подобно `cont`:

```haskell
shift :: ((a -> r) -> Cont r r) -> Cont r a
shift f = Cont \k -> runCont (f k) id
```

`reset` же в качестве продолжения вычислению-аргументу передаёт `id`, тем самым это вычисление не имеет доступа к продолжению после `reset` (в то время как продолжение самого Haskell запоминает, что после `comp` нужно исполнить `k`):

```haskell
reset :: Cont a a -> Cont r a
reset comp = Cont \k -> k (runCont comp id)
```

Упражнение 50 *Каким будет результат исполнения следующей функции:*

```haskell
exampleShiftReset = flip runCont id $
  (1 +) <$> reset ((2 +) <$> shift \k -> pure (k 3 + k 5))
```

6.4 Приложения продолжений

Продолжения полезны для понимания смысла программ, которые мы пишем каждый день (см. 107). Кроме того, как мы увидим в этой главе, продолжения первого класса можно использовать как средство построения могущественных встроенных языков.

6.4.1 Всё через продолжения

Реализуем ряд полезных встроенных языков с помощью продолжений первого класса. Чтобы получить доступ к продолжениям, воспользуемся shallow embedded языком `Cont`[155]. Таким образом, будем работать с башней языков Haskell, `Cont`, X, где X — рассматриваемый встроенный язык.

Начнём с тривиального встроенного императивного языка:

```haskell
runIdentityC :: (forall r . Cont r a) -> a
runIdentityC comp = runCont comp id

exampleIdentity :: Int
exampleIdentity = runIdentityC do
  x <- pure 4
  y <- pure 5
  pure (x + y)
```

Реализуем язык с исключениями. Для этого модифицируем первый типовый параметр `Cont`, **answer (response) type**[156]. Функция `abort` игнорирует продолжение программы до `runExn`, которая завершает продолжение оборачиванием успешного результата в `Just`:

[155] https://blog.poisson.chat/posts/2019-10-26-reasonable-continuations.html
[156] https://wiki.haskell.org/Cont_computations_as_question-answering_boxes

```haskell
abort :: Cont (Maybe r) a
abort = Cont $ const Nothing

runExn :: (forall r . Cont (Maybe r) a) -> Maybe a
runExn comp = runCont comp Just

exampleExn :: Int -> Maybe Int
exampleExn n = runExn do
  when (n < 0) abort
  pure (n + 1)
```

Обратите внимание, что семантика операции обрыва вычисления полностью сосредоточена в функции `abort`, в то время как в `Monad Maybe` соответствующая функция бы отвечала лишь за конструирование элемента домена, а уже монадическое связывание — за работу с продолжениями. Имея доступ к first-class continuations, нам уже не нужно реализовывать монадическое связывание вручную. Говорят, что для произвольного `m`, `Cont (m r) a` — "бесплатная" реализация монады[157].

Реализация работы с ошибками через возвращение специального результата (`Either` в Haskell, `Result` в Rust, `nil` в Go...) имеет монадическую семантику (условные ветвления на каждом выбрасывают только часть продолжения), в то время как полноценный механизм исключений имеет доступ ко всему ограниченному продолжению до соответствующего блока `try-catch`, что напоминает нашу текущую реализацию. С точки зрения количества синтаксического шума на уровне исходного кода, вариант с исключениями явно предпочтительнее, а вопрос типизации исключений мы рассмотрим далее (см. **??**). С точки зрения производительности всё не так очевидно[158][159], однако stack unwinding средствами рантайма как будто имеет большее пространство для оптимизаций.

Теперь попробуем вызвать продолжение несколько раз. Получим язык с недетерминизмом (или backtracking). Так, `choice` дважды продолжает остаток программы и аккумулирует все полученные результаты в списке:

```haskell
choice :: Cont [r] Bool
choice = Cont \k -> k True ++ k False

runNondet :: (forall r . Cont [r] a) -> [a]
runNondet comp = runCont comp (:[])

exampleNondet :: Int -> [Int]
exampleNondet n = runNondet do
  b <- choice
  if b then pure n else pure (n - 1)
```

[157] https://hackage.haskell.org/package/kan-extensions-5.2/docs/Control-Monad-Codensity.html
[158] https://www.serpentine.com/2011/02/25/cps-is-great-cps-is-terrible/
[159] https://stackoverflow.com/questions/13835817/are-exceptions-in-c-really-slow

Удивительно, но имея только продолжения первого класса можно реализовать даже язык с изменяемой ячейкой памяти:

```haskell
-- s -> (s, a) ~ ((s, a) -> r) -> (s -> r)
-- ~ (a -> s -> r) -> (s -> r) ~ Cont (s -> r) a

get :: Cont (s -> r) s -- (s -> s -> r) -> (s -> r)
get = Cont \k s -> k s s

put :: s -> Cont (s -> r) ()
put s' = Cont \k _s -> k () s'

runStateC :: (forall r . Cont (s -> r) a) -> s -> (a, s)
runStateC comp = runCont comp (,)

exampleState :: Int -> ((), Int)
exampleState = runStateC do
  s <- get
  put (s + 1)
```

Идея довольно простая: захват продолжения оставляет после себя функцию, в которую предыдущий вызов продолжения подкладывает текущее состояние. И действительно, изменяемое состояние это просто фрагмент продолжения (например, аппаратного стека), к которому мы имеем непосредственный доступ на чтение и изменение. Здесь же мы разместили в начале продолжения аппликацию, к которой, захватывая продолжение, прыгаем, чтобы получить аргумент и обновить её на новую.

Упражнение 51 *Поредуцируйте пример выше, чтобы понять, как это работает.*

Интуитивно могущество продолжений можно попробовать объяснить следующим образом: поддерживать продолжение — это основная обязанность системы исполнения языка; когда продолжения передаются пользователю, его код как бы вовлекается в деятельность runtime'а и становится его частью.

6.4.2 The mother of all monads

В предыдущем параграфе (6.4.1) мы реализовали возможности классических монад в языке с продолжениями. Сработает ли это для произвольной монады? Оказывается, что да[160][161] Filinski [1994].

[160] http://www.schoolofhaskell.com/school/to-infinity-and-beyond/pick-of-the-week/the-mother-of-all-monads

[161] https://blog.poisson.chat/posts/2019-10-26-reasonable-continuations.html

Вспомним, что монада — это конструктор типа `m`, которым мы представляем денотацию вычисления. Операции `pure` и `>>=` позволяют создавать и композировать денотации. Таким образом, мы имеем вычисления, представленные в виде first-class данных.

Если наш язык предоставляет продолжения первого класса (для примера возьмём `Cont (m r)` как такой язык), мы можем определить две операции: `reflect` — исполнить монадическое вычисление в языке с продолжениями и получить результат `a`; `reify` — по вычислению в языке с продолжениями получить денотацию[162]. В Haskell можно это выразить как перегрузку этой пары функций для различных монад:

```
class MonadicReflection m where
  -- m a -> (a -> m r) -> m r
  reflect :: m a -> forall r . Cont (m r) a
  -- (forall r . (a -> m r) -> m r) -> m a
  reify :: (forall r . Cont (m r) a) -> m a
```

Например, для домена `State` реализация будет выглядеть следующим образом:

```
newtype State s a = State { runState :: s -> (a, s) }

instance MonadicReflection (State s) where
  reflect :: State s a -> (forall r . Cont (State s r) a)
  reflect comp = Cont \k -> State \s ->
    let (a, s') = runState comp s in
    runState (k a) s'

  reify :: (forall r . Cont (State s r) a) -> State s a
  reify comp = State $ runState (runCont comp (\x -> State (x,)))

exampleStateReflection :: Int -> (Int, Int)
exampleStateReflection = runState $ reify do
  x <- reflect get
  reflect $ put (x + 1)
  pure x
```

Интуитивно это можно понимать следующим образом: реализации bind на самом деле не важно, ей подаётся “маленькое” продолжение (собранное с помощью рассахаривания `do`-нотации), или “большое” продолжение, собранное мета-языком или монадой `Cont`. И в принципе монады можно задавать через их вложение в монаду `Cont`[163].

Таким образом, если язык поддерживает продолжения первого класса, можно в direct-стиле[164] реализовать функциональность произвольной монады, без `do`-нотации, аппликатив-

[162] Для энергичного языка `reify` должна принимать thunk `(() -> a) -> m a`.

[163] https://blog.poisson.chat/posts/2019-10-27-continuation-submonads.html

[164] https://www.unison-lang.org/docs/fundamentals/abilities/for-monadically-inclined/

ных цепочек и прочего синтаксического шума[165][166]. Позже мы увидим, что этот подход, в отличие от монад, нативно поддерживает композицию (7), а также не накладывает ограничений на типизацию (**??**). Отсюда возникает вопрос: а нужны ли нам теперь в программировании монады?

6.4.3 Генераторы и корутины

Генераторы и корутины обычно определяют как вычисление, которое может быть остановлено и возобновлено снова в том же состоянии Moura and Ierusalimschy [2009]. Корутины же обобщают генераторы и, как правило, используются как примитив асинхронного программирования в виде `async/await` или других языковых конструкций Elizarov et al. [2021]. Различие синхронного и асинхронного программирования можно понимать так: в первом случае продолжениями управляет исключительно операционная система, а во втором — средства языка.

Для примера реализуем генераторы. В качестве результата генератора будем использовать ленивый список, итератор, закодированный в виде развёртки (см. 4.5):

```haskell
data Box f = forall s . Box s (s -> f s)
data ListF a rec = Nil | Cons a rec
type Iterator a = Box (ListF a)

box2list :: Iterator a -> [a]
box2list (Box s next) = case next s of
  Nil -> []
  Cons x s' -> x : box2list (Box s' next)
```

Скрытым состоянием итератора будет ещё невыполненное продолжение. Доступ к продолжению будем получать в монаде `Cont`, где в качестве response type возьмём `GenState`. Так, операция `yield` останавливает вычисление и сохраняет продолжение в конструкторе `Yield`, а `makeGen` заканчивает данный генератор порождением конструктора `Stop`:

```haskell
data GenState a = Stop | Yield a (() -> GenState a)

makeGen :: Cont (GenState a) () -> Iterator a
makeGen comp = Box
  (\() -> runCont comp (\() -> Stop))
  (\k -> case k () of
      Stop -> Nil
      Yield x k' -> Cons x k'
  )
```

[165] http://www.valuedlessons.com/2008/01/monads-in-python-with-nice-syntax.html
[166] https://github.com/lampepfl/monadic-reflection/blob/main/TUTORIAL.md

```haskell
yield :: a -> Cont (GenState a) ()
yield x = cont \k -> Yield x k
```

Теперь мы можем писать код, последовательно порождающий множество результатов:

```haskell
exampleGen :: [Int]
exampleGen = it2list $ makeGen do
  yield 1
  yield 2
  yield 3
```

Упражнение 52 *Приведите вычисление* `runCont exampleGen (\() -> Stop))` *к нормальной форме.*

6.5 Эффективная работа с продолжениями

На практике для реализации генераторов и корутин требуется останавливать и продолжать программу, иначе говоря, захватывать продолжения. Это нужно делать максимально эффективно.

6.5.1 Contiguous stack

Продолжение представляется в виде аппаратного стека. Как только требуется захватить продолжение, стек копируется в кучу (возможно, лениво)[167]. То есть этот подход полностью полагается на поддержку со стороны рантайма языка.

6.5.2 Сегментный стек

Стек вызовов представляется как связный список аллоцированных в куче сегментов, каждый соответствует ограничивающей операции. Так, не требуется делать копирования, достаточно подмены указателей. Однако, в таком случае стек нелокален, что не очень хорошо для работы кешей.

6.5.3 Finite state machine (FSM)

Данная реализация подразумевает автоматическую CPS трансформацию пользовательского кода средствами компилятора. Чтобы не аллоцировать большое количество замыканий, продолжения дефункционализируются и в рамках каждой функции представляются одним изменяемым объектом. Так, состояние функции целиком один раз аллоцируется в куче, а возобновление кода тела функции в определённом месте реализовано как машина состояний — с помощью меток и прыжков[168].

[167] (youtube) Иван Углянский - Java Project Loom.

[168] `https://github.com/Kotlin/KEEP/blob/master/proposals/coroutines.md#state-machines`

Таким образом, например, реализованы корутины в Kotlin[168], генераторы в C#[169]...

Даже в таком виде CPS остаётся тяжеловесной трансформацией, способной замедлить исполнение кода на порядки. Дело, в частности, в том, что переменные в таком подходе сложно размещать в регистрах (у функций много точек выходов и входов[168]), приходится постоянно записывать их в RAM — производить **spilling**[170].

[169] https://csharpindepth.com/Articles/IteratorBlockImplementation
[170] https://en.wikipedia.org/wiki/Register_allocation

7 Эффекты и модульные интерпретаторы

Ранее мы признали работу со сложностью главной задачей программиста, а построение встроенных языков — основным инструментом её решения (5.1). В данной главе мы рассмотрим понятие эффекта. Оно тесно связано со встроенными языками и даст нам лучшее понимание, когда их конструировать, что это даёт, и с чем нужно быть осторожным.

Для реализации встроенных языков мы предпочли shallow embedding в форме tagless final (5.4), который максимально переиспользует возможности мета-языка и позволяет давать различные интерпретации одной программе. Далее мы исследовали процесс вычисления и извлекли понятие продолжения (6). Оказалось, что tagless final языки, которые мы строили вокруг монад, можно выразить через продолжения удобнее и проще (6.4.1, 6.4.2). В этой главе мы поймём, как это поможет решить expression problem (5.5) до конца.

С историческо-философской перспективы, теория эффектов является прочным мостом между функциональной и императивной парадигмами программирования. Они возникли порознь как Машина Тьюринга и λ-исчисление Чёрча, и оставались довольно изолированными школами мысли довольно долгое время. Это стало меняться в последние 30 лет, и мы стали понимать, как эти два мира дополняют друг друга. Теория эффектов до сих пор является крайне горячей темой[171][172].

7.1 Понятие эффекта

Начнём разговор от обратного, со свойства чистоты. **Чистая функция** обладает следующими свойствами:

- Её результат всегда одинаков при одинаковом наборе аргументов (никак более нетривиально не зависит ни от чего более);
- Её единственный наблюдаемый результат — её возвращаемое значение.

В целом стиль программирования с использованием чистых функций приветствуется, так как он обладает множеством хороших свойств. Так, про них можно удобно рассуждать с помощью equational reasoning; всё, что нужно для понимания кода, явно написано в этом коде; классические системы типов хорошо работают, предоставляя полноту абстракции, качественную документацию и частичную спецификацию... Также известно, что всё можно записать с помощью чистых вычислений, даже работу с IO Jones [2001].

Однако, используя только чистые функции, всё приходится делать вручную. В случае с IO (с состоянием аналогично) — передавать результирующий мир в аргументы раз за разом:

```
1 getList :: Int -> World -> (World, [Int])
2 getList n w | n == 0 = (w, [])
3             | otherwise =
4   let (w', x) = getInt w in
```

[171] (youtube) The Evolution of Effects — Nickolas Wu.

[172] https://github.com/yallop/effects-bibliography

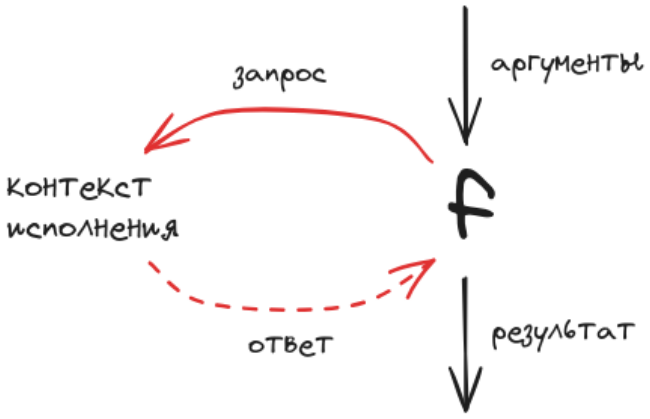


Рис. 25: Эффекты как клиент-серверное взаимодействие.

```
5     let (w'', xs) = getList (n - 1) w' in
6     (w'', x : xs)
```

Таким образом, код из чистых функций заполнен несущественными церемониями, за которыми не видно бизнес-логики и сути. Чтобы сосредоточиться на важных деталях, нужно делегировать весь этот bookkeeping стороннему коду, замести неинтересные детали под ковёр. Тогда код выше можно будет переписать, например, следующим образом:

```
1  getList :: Int -> IO [Int]
2  getList n | n == 0 = pure []
3            | otherwise = do
4    x <- getInt
5    xs <- getList (n - 1)
6    return (x : xs)
```

Абстрактную сущность, которой мы будем делегировать несущественные для данного фрагмента бизнес-логики детали, мы будем называть **контекстом исполнения (execution context)**[173]. А **эффектом** — взаимодействие функции с контекстом исполнения, которое происходит с помощью вызова **effect-операций** (например, `getInt` из примера выше)[174]. Так, на контекст исполнения можно смотреть как на сервер, которому функция-клиент шлёт запросы и получает ответы (см. рис. 25). В этой модели такая функция может нарушать оба свойства чистых функций.

На практике этим контекстом является интерпретатор (встроенного) языка, а effect-операциями — его конструкции. Если язык является встроенным, то говорят о **пользовательских (user-defined) эффектах**. Так мы снова возвращаемся к задаче построения модульных интерпретаторов (5.5). К тому же, реализация вычислительного контекста также может делегировать реализацию некоторой функциональности другому контексту исполнения, и так далее. Получаем уже знакомую нам башню интерпретаторов (5.1.1).

[173] `https://okmij.org/ftp/Computation/having-effect.html`

[174] Также эффектом иногда называют функцию перехода в dataflow-анализе Møller and Schwartzbach [2012].

Рассмотрим некоторые примеры вычислительных контекстов и операций:

- Контекст — подсистема управления памятью, `modify` — effect-операция: контекст для нас поддерживает состояние ячеек памяти;
- Контекст — хендлер исключения, `throw MyException` — effect-операция: контекст за нас определяет, как ошибка будет обрабатываться (обратите внимание, что тут управление не возвращается терму);
- Контекст — настройки инъекции зависимостей, запрос функциональности — effect-операция: контекст за нас определяет реализацию функциональности, которой нам пользоваться[175]. Как мы увидим далее, этот и подобные простые эффекты можно реализовать просто и эффективно (см. 7.2.2).

Упражнение 53 *Приведите ещё примеры вычислительных контекстов и операций.*

Когда мы говорим про standalone язык, на котором мы программируем (например, Haskell), любое наше действие в программе исполняется им. То есть, например, сложение — effect-операция? В таком случае разумно выделить подмножество конструкций языка, достаточно "интересных", чтобы считать, что они порождают эффект.

Какие конструкции языка считать "интересными"? Заметим, что с одной стороны это хорошо, что эффекты скрывают от нас некоторую сложность, позволяя сосредоточиться на других вещах. С другой стороны, это же и плохо, ведь мы эту сложность перестаём наблюдать, а она пронизывает наш код, поддерживает неявные зависимости между его частями. Таким образом, эффекты требуют дополнительной аккуратности со стороны программиста[176]. Соответственно, именно такие непростые конструкции и стоит считать "интересными". Как минимум точно стоит считать "интересными" конструкции, использование которых выводит функцию из категории чистых. Также, это могут быть операции, делающие сложные нелокальные модификации потока управления (поддерживаемого интерпретатором в виде продолжения).

В конечном итоге выбор "интересных" конструкций зависит от задачи и перспективы разработчика[173]. Так, конструкции, влияющие на произвольные наблюдаемые свойства кода, как, например, терминируемость или вычислительная сложность, могут мотивировать считать рекурсивные вызовы или долгие операции эффектами.

Далее мы научимся отслеживать и контролировать использование эффектов на уровне типов с помощью систем эффектов (см. далее **??**).

7.2 Хендлеры эффектов

Хендлеры эффектов — это современный универсальный метод построения модульных интерпретаторов встроенных языков, напрямую реализующий клиент-серверную метафору. Как

[175] Существует термин **contextual polymorphism** — код в разных контекстах может иметь различное поведение.
[176] (youtube) Kris Jenkins — Side-Effects Are The Complexity Iceberg.

обычно бывает, хендлеры были изобретены множество раз. В этой главе мы посмотрим на основные реализации, которые лучше всего помогут нам понять концепцию.

Основная идея хендлеров эффектов довольно проста. Вводится языковая конструкция `handle`, позволяющая задать вычислительный контекст для определённого скоупа, предоставляющий реализации effect-операций. Также вводится конструкция `perform`, позволяющая вызвать effect-операцию (отправить запрос контексту). Каждая операция имеет набор параметров, а также "обратный адрес", продолжение места вызова, в который она вернёт результат. Например, на экспериментальном языке Koka[177] контекст, предоставляющий некоторую константу, может быть реализован следующим образом (`resume` — имя продолжения места вызова, `perform` вставляется неявно):

```
with handler
  ctl ask() resume(21)
ask() + ask()
```

Если ближайший контекст нужный запрос обработать не может, запрос делегируется внешнему контексту, и так пока подходящий контекст не будет найден. На этой идее основывается модульность интерпретаторов, заданных хендлерами.

7.2.1 Хендлеры через ограниченные продолжения

Как мы уже видели ранее, различные эффекты можно реализовывать с помощью доступа к текущему продолжению (6.4.1, 6.4.2). Хендлеры эффектов дополняют эту идею тем, что используют ограниченные продолжения, чтобы передавать управление различным интерпретаторам (хендлерам).

Известно, что классические операторы манипуляции ограниченными продолжениями, monadic reflection и хендлеры эффектов выразимы друг через друга Forster et al. [2017].

7.2.2 Эффективная реализация хендлеров

В общем виде скорость работы `perform` определяется скоростью захвата и восстановления ограниченных продолжений. Однако, существует класс операций, которые можно реализовать гораздо эффективнее.

Если мы посмотрим на реализацию операции `ask`, то мы увидим, что она последним действием вызывает продолжение, возвращая управление вызвавшему коду. Такие операции называют **tail-resumptive**, они очень сильно напоминают обычные функции, за исключением того, что их реализации определяются контекстом (хендлером). Таким образом, tail-resumptive операции можно реализовать как неявную передачу словаря функций от хендлера к `perform`, и тем самым избежать дорогих манипуляций продолжениями Xie et al. [2020][178].

[177] `https://koka-lang.github.io/koka/doc/index.html`

[178] Хендлеры tail-resumptive операций напоминают co-pattern-matching и, соответственно, объекты (см. 4.6.2).

7.2.3 Встроенные хендлеры как явная клиент-серверная коммуникация

Чтобы лучшим образом понять семантику хендлеров, реализуем язык с хендлерами как встроенный в Haskell. Начнём с варианта, предложенного Олегом Киселёвым, максимально прямолинейно кодирующего идею клиент-серверной коммуникации терма и контекста Kiselyov et al. [2013].

Начнём с эффекта `ask`, запрашивающего числа у контекста. Зададим тип данных сообщений к контексту, это либо конечный результат вычисления, либо запрос `Ask`, содержащий "обратный адрес" — текущее продолжение:

```
data Message res = Val res | Ask (Int -> Message res)
```

Продолжения будем собирать в специализированный `Monad Cont` с подходящим response type (см. 6.2.2, 6.4.1):

```
newtype Eff res = Eff
  { runEff :: forall res' . (res -> Message res') -> Message res' }
```

Тогда эффект `ask` реализуется просто как "отправка" запроса `Ask` с текущим продолжением:[179]

```
ask :: Eff Int
ask = Eff \k -> Ask k
```

Хендлер мы реализуем как "сервер", который в цикле обрабатывает запросы, пока вычисление не пришлёт конечный результат:

```
run :: Eff res -> Message res
run comp = runEff comp Val

runReader :: Eff res -> Int -> res
runReader comp env = loop (run comp)
  where
    loop = \case
      Val res -> res
      Ask k -> loop (k env)
```

Наконец, мы можем писать effectful код:

```
exampleReader :: Int -> Int
exampleReader = runReader do
  x <- ask
  y <- ask
  pure (x + y)
```

[179] Ранее мы аналогично реализовывали генераторы, см. 6.4.3.

Монада `Eff` превратит его в ленивый список сообщений:

```
exampleReader :: Int -> Int
exampleReader = loop $
  Ask \x ->
  Ask \y ->
  Val (x + y)
```

7.2.4 Расширяемые сообщения и пересылка

Абстрагируем тип сообщений по "форме" запросов, которые в них могут участвовать (см. 4.3.2):

```
data Message effs res = Val res | Request (effs (Message effs res))
```

Предыдущий тип сообщений получается передачей следующего функтора формы:

```
newtype Reader env msg = Ask (env -> msg)
```

Упражнение 54 *Убедитесь, что* `Message (Reader Int) res` *эквивалентно предыдущему типу сообщений.*

Копроизведение функторов формы является функтором формы (см. 5.5.1):

```
data (eff |> effs) a = L (eff a) | R (effs a)
```

Теперь операция `ask` допускает существование других типов запросов `effs`:

```
ask :: Eff (Reader env |> effs) env
ask = Eff \k -> Msg $ L $ Ask k
```

Новый хендлер обрабатывает только часть запросов, остальные пересылает хендлеру выше (скомпозировав правильным образом продолжения):

```
runReader
  :: forall effs env res . Functor es
  => Eff (Reader env |> effs) res
  -> env -> Eff effs res
runReader comp env = loop (run comp)
  where
    loop :: Request (Reader env |> effs) res -> Eff effs res
    loop = \case
      Val res -> pure res
      Msg (L (Ask k)) -> loop (k env)
      Msg (R unknownReq) -> do
        response <- Eff \k -> Msg (fmap k unknownReq)
        loop response
```

Заметьте, что в результирующем домене остались непроинтерпретированные эффекты. Конечный домен получится применением всех необходимых интерпретаторов. Так решается stable denotations problem (5.5).

7.2.5 Свободные монады

Чтобы получить полноценное решение более простым и минималистичным образом, перейдём к самой классической реализации хендлеров через свободные монады. Для начала обсудим сами свободные монады.

Рассмотрим некоторую алгебраическую структуру, например, моноид (нейтральный элемент и ассоциативная бинарная операция). По произвольному множеству X можно построить некоторый моноид $M(X)$ наиболее "экономичным" образом — **свободный моноид**. Это делается следующим образом: к множеству X добавляют деревья выражений с операциями моноида:

```
data M x = Element x | Mempty | Mappend (M x) (M x)
instance Monoid (M x) where
  mempty = Mempty
  mappend l r = Mappend l r
```

Ещё только нужно организовать новое множество таким образом, чтобы в нём не было одинаковых с точки зрения алгебры деревьев (например, `Mappend Mempty Mempty` и `Mempty`). В случае моноида в этом случае можно выбрать тип списка:

```
type M x = [x]
instance Monoid (M x) where
  mempty = []
  mappend = (++)
```

Свободные монады строятся аналогично свободным моноидам. Используем определение монады как функтора с операциями `pure` и `join`. Тогда по произвольному функтору F можно получить монаду $Free(F)$ (покажем это с помощью эквивалентного определения монад из Haskell):

```
data Free f a = Pure a | Join (f (Free f a))
instance Functor f => Monad (Free f) where
  return = Pure
  Pure x >>= k = k x
  Join f >>= k = Join (fmap (>>= k) f)
```

Переименуем конструкторы:

```
data Term sig var = Var var | Op (sig (Term sig var))
(>>=) :: Term sig var -> (var -> Term sig var') -> Term sig var'
data MonoidSig subtree = Mempty | MAppend subtree subtree -- (Bool -> subtree)
```

Это не что иное, как кодирование алгебраических термов над сигнатурой `sig` и переменными из множества `var`. А монадическое связывание — это подстановка.

7.2.6 Хендлеры через свободные монады

Заметим, что следующие типы изоморфны: `Message` ≅ `Free` ≅ `Term`. Хендлеры в классическом виде как раз и возникли в процессе изучения того, как описывать эффекты в виде алгебраических структур Bauer [2018][180].

Вместо переменных мы будем хранить чистый результат вычисления, а сигнатуры будем записывать в виде $P \times (A \to K)$, где P — параметр операции, а A — результат операции, по которому хендлер выбирает нужный подтерм-продолжение для возобновления вычисления:

```
data Comp effs res = Res res | Op (effs (Comp effs res))
data Reader env comp = Ask () (env -> comp)
data State s comp = Get () (s -> comp) | Put s (() -> comp)
```

Реализация `Monad Comp` будет как раз композировать продолжения для нас в `do`-нотации:

```
ask = Op (Ask () (\e -> Val e)
example = do
  x <- ask
  y <- ask
  pure (x + y)
-- αβ-эквивалентно
example =
  Op (Ask () (\x ->
  Op (Ask () (\y ->
  Res (x + y)))))
```

Хендлер также просто сворачивает список операций. Операции, которые он не умеет обрабатывать, он оставляет в дереве. Чтобы пропустить неизвестную операцию и интерпретировать поддерево (продолжение), используется `fmap`:

```
runReader
  :: Functor effs => Comp (Reader env |> effs) res -> env -> Comp effs res
runReader comp env = case comp of
  Res res -> Res res
  Op (L (Ask () k)) -> runReader (k env) env
  Op (R other) -> Op (fmap (`runReader` env) other)
```

Упражнение 55 *Постройте пример вычисления из* `Reader` *и* `State`*. Какое дерево получится после интерпретации одного из эффектов?*

Как обычно, свёртку можно обобщить в виде катаморфизма:

[180](youtube) What is algebraic about algebraic effects and handlers — Andrej Bauer.

```
handle
  :: Functor effs => (res -> d) -> (effs d -> d) -> Comp effs res -> d
handle val alg = \case
  Pure res -> val res
  Op eff -> alg $ fmap (handle val alg) eff
```

Теперь в реализации продолжение (поддерево) уже проинтерпретировано в нужный домен:

```
runReader
  :: Functor effs => Eff (Reader env |> effs) res -> env -> Eff effs res
runReader = handle (\res _env -> pure res) \case
  L (Ask k) -> \env -> k env env
  R other -> \env -> Op (fmap ($ env) other)
```

Когда хендлеры реализуют как built-in возможность в языке, нужно принять дизайн-решение, в предоставляемом продолжении текущий эффект уже проинтерпретирован, или нет. Первый вариант называют **deep handlers**, второй — **shallow handlers**, они выразимы друг через друга Hillerström and Lindley [2018].

К сожалению, поскольку различные варианты эффектов упорядочены как на уровне термов, так и на уровне типов, нам нужна операция, превращающая вычисление от меньшего количества эффектов в вычисление с большим:

```
liftF :: Functor effs => Comp effs res -> Comp (eff |> effs) res
liftF = \case
  Pure x -> Pure x
  Op effs -> Op $ R $ liftF <$> effs

example :: Comp (Reader Int |> State Int) ()
example = do
  env <- ask
  liftF (put env)
```

От порядка можно избавиться с помощью классов типов Swierstra [2008] (3.4.8).

Можно заметить, что помимо большого количества аллокаций, использование `Monad Free` может приводить к квадратичной сложности кода из-за линейных проходов в каждом bind'е. Существуют различные альтернативные схемы кодирования Ploeg and Kiselyov [2014], Kiselyov and Ishii [2015].

Свободные монады находят и другие, правда, аналогичные применения: trampolining Bjarnarson [2012] и пайплайны Kiselyov [2012b] [Bragilevsky, chapter 14].

7.2.7 Приложения хендлеров

Рассмотрим хендлеры tail-resumptive операций, обращающиеся с продолжениями тривиальным образом. Они нужны для распространения значений и функциональности вниз по

стеку. Как мы обсуждали ранее 7.2.2, такие хендлеры аналогичны наличию динамических свободных переменных или неявных аргументов функций 3.1.2, и, вместе с ними, рекордов или анонимных классов 4.6.2.

Среди нетривиальных сценариев использования продолжений выделим следующие:

- Продолжение можно не вызвать, так, можно реализовать механизм исключений. Однако, на практике продолжения могут содержать логику финализации ресурсов. В этом случае всё равно нужна какая-то специальная обработка [181].
- Можно вызвать несколько раз для эмуляции недетерминизма. Это ограничивает использование эффективных мутабельных продолжений 6.5 Leijen [2018][182], а также требует нетривиальной обработки при работе с ресурсами.
- Можно вызвать не сразу. Это нужно для реализации изменяемого состояния, генераторов и корутин. Для изменяемого состояния это слишком дорого. Генераторы — хорошее применение, но built-in реализация генераторов может быть эффективнее за счёт экономии аллокаций продолжений (создаётся сразу `Iterator`)[183]. Реализация корутин поверх хендлеров даёт возможность пользователям писать собственные планировщики, что самое котируемое применение хендлеров на данный момент Sivaramakrishnan et al. [2021], Phipps-Costin et al. [2023].

7.2.8 Трансформеры монад

В классической реализации хендлеров через свободные монады 7.2.6, каждый хендлер порождает промежуточное дерево из непроинтерпретированных операций. Естественным образом возникает желание дефорестировать 4.4.1 эти промежуточные деревья. Можно пойти до конца и полностью избавиться от свободных монад и функторов сигнатур Wu and Schrijvers [2015]. Так, мы фактически получим tagless final shallow embedding 5.4, другое популярное решение expression (stable denotations) problem — **трансформеры монад** Liang et al. [1995], Jones [1995][184]. В целом эти подходы, равносильны по выразительности, однако могут иметь на практике различные особенности встраивания в язык Schrijvers et al. [2019]. Однако, в отличие от трансформеров, хендлеры можно сделать удобной built-in возможностью языка.

7.2.9 Алгебраичность и эффекты высших порядков

В начале нулевых появилась идея описывать effect-операции не монадами сразу, а алгебраически, с помощью сигнатур и уравнений Plotkin and Power [2002], Bauer [2018]. Эффекты в этом формализме являются композируемыми по построению, как композируемы сигнатуры алгебраических теорий (конкатенация сигнатур — сигнатура 7.2.4), так мы имеем расширяемый синтаксис. Однако, на операции накладывается ограничение в виде **свойства**

[181] `https://koka-lang.github.io/koka/doc/book.html#sec-resource`
[182] `https://koka-lang.github.io/koka/doc/book.html#sec-multi-resume`
[183] `https://csharpindepth.com/Articles/IteratorBlockImplementation`
[184] `https://hackage.haskell.org/package/mtl`

алгебраичности — операция коммутирует с продолжением и “всплывает” наверх:

$$E[op(v, k)] \equiv op(v, \lambda x.\, E[k(x)])$$

Соответствующие эффекты называют **алгебраическими**.

В нашей реализации (7.2.6) ограничение на алгебраичность можно увидеть следующим образом. У нас сигнатуры операций являются функторами по своим продолжениям. Этим пользуются как монадическое связывание для накапливания продолжений, так и хендлеры для интерпретации продолжений неизвестных операций.

Многие полезные эффекты являются алгебраическими, но не все. Например, поимка исключений `catch` не является алгебраической операцией, так как её сигнатура запишется следующим образом:

```
data Catch e comp = Catch { try :: comp, onExn :: (e -> comp), next :: comp }
```

В таком случае единственная разумная реализация функтора будет работать, в том числе, и с вложенными продолжениями, и вся программа, в результате работы bind, окажется внутри блока `try`.

Иначе говоря, *алгебраические эффекты не могут принимать другие effectful вычисления в качестве аргументов*. Как раз, чтобы моделировать `catch`, и были оригинально предложены хендлеры Plotkin and Pretnar [2013][185]. Таким образом, у хендлеров две задачи: ограничивать скоуп некоторой функциональности и интерпретировать эффекты. Однако, как мы знаем, порядок хендлеров определяет результирующий домен и, соответственно, семантику. В то же время задача ограничения скоупа фиксирует позицию хендлера, что ограничивает выразительность и делает некоторые домены недоступными.

Операции, не удовлетворяющие свойству алгебраичности, соответствуют **эффектам высших порядков (higher-order effects)**, которы могут принимать другие effectful вычисления в качестве аргументов. Существуют встраивания хендлеров таких эффектов и обширный набор исследований на тему Wu et al. [2014][186][187] Yang et al. [2022].

Ключ к эффектам высших порядков состоит в возможности исполнять эффекты вычислений-аргументов в контексте тех хендлеров, которые доступны на колсайте эффекта высших порядков van der Rest et al. [2022]. Это связано с другой идеей, bidirectional effects, позволяющей реализации операции порождать эффекты на своём колсайте (например, бросать там исключения), что крайне необходимая на практике возможность Zhang et al. [2020][188].

To be continued...

[185] Несмотря на то, что алгебраические эффекты — это просто класс “хороших” операций, часто при их упоминании подразумевают хендлеры эффектов.

[186] `https://github.com/fused-effects/fused-effects`

[187] (youtube) Building Haskell Programs with Fused Effects — Patrick Thomson

[188] `https://effekt-lang.org/docs/concepts/bidirectional`

Список литературы

Christopher Strachey. Fundamental concepts in programming languages. *Higher-order and symbolic computation*, 13:11–49, 2000. URL https://facweb.cdm.depaul.edu/smitsch/courses/csc447fa23/assets/articles/strachey-fundamental-concepts-in-programming-languages.pdf.

Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys (CSUR)*, 17(4):471–523, 1985. URL https://doi.org/10.1145/6041.6042.

Benjamin C Pierce. *Types and programming languages*. MIT press, 2002.

John Launchbury and Simon L Peyton Jones. State in haskell. *Lisp and symbolic computation*, 8(4): 293–341, 1995. URL https://www.microsoft.com/en-us/research/wp-content/uploads/2016/07/state-lasc.pdf.

Sandy Maguire. *Thinking with Types, Type-Level Programming in Haskell*. a. URL https://leanpub.com/thinking-with-types/?

Simon Peyton Jones. Type inference as constraint solving: how ghc's type inference engine actually works. Keynote talk at Zurihac 2019, 2019.

Benjamin C Pierce and David N Turner. Local type inference. *Acm transactions on programming languages and systems (toplas)*, 22(1):1–44, 2000. URL https://doi.org/10.1145/345099.345100.

David Raymond Christiansen. Bidirectional typing rules: A tutorial. 2013. URL https://davidchristiansen.dk/tutorials/bidirectional.pdf.

Jana Dunfield and Neelakantan R Krishnaswami. Sound and complete bidirectional typechecking for higher-rank polymorphism with existentials and indexed types. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–28, 2019. URL https://doi.org/10.1145/3290322.

Alejandro Serrano, Jurriaan Hage, Simon Peyton Jones, and Dimitrios Vytiniotis. A quick look at impredicativity. *Proceedings of the ACM on Programming Languages*, 4 (ICFP):1–29, 2020. URL https://www.microsoft.com/en-us/research/publication/a-quick-look-at-impredicativity/.

Frank Emrich, Sam Lindley, Jan Stolarek, James Cheney, and Jonathan Coates. Freezeml: Complete and easy type inference for first-class polymorphism. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 423–437, 2020. URL https://link.springer.com/article/10.1208/s12249-010-9382-3.

Brent A Yorgey, Stephanie Weirich, Julien Cretin, Simon Peyton Jones, Dimitrios Vytiniotis, and José Pedro Magalhães. Giving haskell a promotion. In *Proceedings of the 8th ACM SIGPLAN Workshop on Types in Language Design and Implementation*, pages 53–66, 2012. URL https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/p53-yorgey.pdf.

Stephanie Weirich, Justin Hsu, and Richard A Eisenberg. System fc with explicit kind equality. *ACM SIGPLAN Notices*, 48(9):275–286, 2013. URL https://doi.org/10.1145/2544174.2500599.

Vitaly Bragilevsky. *Haskell in Depth*. Manning. URL https://www.manning.com/books/haskell-in-depth.

Bruno Blanchet. Escape analysis for object-oriented languages: application to java. *Acm Sigplan Notices*, 34(10):20–34, 1999. URL https://doi.org/10.1145/320385.320387.

Richard A Eisenberg and Simon Peyton Jones. Levity polymorphism. *ACM SIGPLAN Notices*, 52 (6):525–539, 2017. URL https://doi.org/10.1145/3140587.3062357.

Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler. A history of haskell: being lazy with class. In *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, pages 12–1, 2007. URL https://www.researchgate.net/profile/Simon-Peyton-Jones/publication/221501761_A_history_of_Haskell_Being_lazy_with_class/links/0c960517e31f50f743000000/A-history-of-Haskell-Being-lazy-with-class.pdf.

Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 60–76, 1989. URL https://doi.org/10.1145/75277.75283.

Cordelia V Hall, Kevin Hammond, Simon L Peyton Jones, and Philip L Wadler. Type classes in haskell. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 18(2):109–138, 1996. URL https://doi.org/10.1145/227699.227700.

Simon Peyton Jones, Mark Jones, and Erik Meijer. Type classes: an exploration of the design space. In *In Haskell Workshop*, 1997. URL https://courses.cs.washington.edu/courses/cse590p/06sp/multi.pdf.

Jeffrey R Lewis, John Launchbury, Erik Meijer, and Mark B Shields. Implicit parameters: Dynamic scoping with static types. In *Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 108–118, 2000. URL https://doi.org/10.1145/325694.325708.

Simon Peyton Jones. Type inference as constraint solving: how ghc's type inference engine actually works, 2019. URL https://www.microsoft.com/en-us/research/publication/type-inference-as-constraint-solving-how-ghcs-type-inference-engine-actually-works/.

Martin Sulzmann, Gregory J Duck, Simon Peyton-Jones, and Peter J Stuckey. Understanding functional dependencies via constraint handling rules. *Journal of functional programming*, 17(1): 83–129, 2007a. URL https://doi.org/10.1017/S0956796806006137.

Oleg Kiselyov and Chung-chieh Shan. Functional pearl: implicit configurations–or, type classes reflect the values of types. In *Proceedings of the 2004 ACM SIGPLAN workshop on Haskell*, pages 33–44, 2004. URL https://d1wqtxts1xzle7.cloudfront.net/43582096/Functional_pearl_implicit_configurations20160310-32037-1bu6179-libre.pdf.

Filip Křikava, Heather Miller, and Jan Vitek. Scala implicits are everywhere: A large-scale study of the use of scala implicits in the wild. *Proceedings of the ACM on Programming Languages*, 3 (OOPSLA):1–28, 2019. URL https://doi.org/10.1145/3360589.

Bruno CdS Oliveira, Adriaan Moors, and Martin Odersky. Type classes as objects and implicits. *ACM Sigplan Notices*, 45(10):341–360, 2010. URL https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=d30d65ca9ce7891352024a5c71ebe0ae8c41f7ac.

Dominique Devriese and Frank Piessens. On the bright side of type classes: instance arguments in agda. *ACM SIGPLAN Notices*, 46(9):143–155, 2011. URL https://archive.alvb.in/msc/thesis/reading/typeclasses-agda_Devriese.pdf.

Conor McBride. Faking it simulating dependent types in haskell. *Journal of functional programming*, 12(4-5):375–392, 2002. URL https://doi.org/10.1017/S0956796802004355.

Simon Peyton Jones, Andrew Tolmach, and Tony Hoare. Playing by the rules: rewriting as a practical optimisation technique in ghc. In *Haskell workshop*, volume 1, pages 203–233, 2001. URL https://www.microsoft.com/en-us/research/wp-content/uploads/2001/09/rules.pdf.

Sandy Maguire. *Algebra-Driven Design, Elegant Solutions from Simple Building Blocks*. b. URL https://leanpub.com/algebra-driven-design/.

Li-yao Xia. *Defunctionalization*. URL https://poisson.chat/aquarium/defunctionalization.pdf.

John C Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of the ACM annual conference-Volume 2*, pages 717–740, 1972. URL https://doi.org/10.1145/800194.805852.

John C Reynolds. Definitional interpreters revisited. *Higher-Order and Symbolic Computation*, 11: 355–361, 1998. URL https://doi.org/10.1023/A:1010075320153.

Jeremy Yallop and Leo White. Lightweight higher-kinded polymorphism. In *International Symposium on Functional and Logic Programming*, pages 119–135. Springer, 2014. URL https://www.cl.cam.ac.uk/~jdy22/papers/lightweight-higher-kinded-polymorphism.pdf.

Manuel MT Chakravarty, Gabriele Keller, and Simon Peyton Jones. Associated type synonyms. *ACM SIGPLAN Notices*, 40(9):241–253, 2005a. URL https://www.microsoft.com/en-us/research/wp-content/uploads/2005/01/at-syns.pdf.

Manuel MT Chakravarty, Gabriele Keller, Simon Peyton Jones, and Simon Marlow. Associated types with class. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–13, 2005b. URL https://www.microsoft.com/en-us/research/wp-content/uploads/2005/01/assoc.pdf.

Tom Schrijvers, Simon Peyton Jones, Manuel Chakravarty, and Martin Sulzmann. Type checking with open type functions. In *Proceedings of the 13th ACM SIGPLAN international conference on Functional programming*, pages 51–62, 2008. URL https://www.microsoft.com/en-us/research/wp-content/uploads/2008/01/icfp2008.pdf.

Richard A Eisenberg, Dimitrios Vytiniotis, Simon Peyton Jones, and Stephanie Weirich. Closed type families with overlapping equations. *ACM SIGPLAN Notices*, 49(1):671–683, 2014. URL https://www.microsoft.com/en-us/research/wp-content/uploads/2016/07/popl137-eisenberg.pdf.

Oleg Kiselyov, Simon Peyton Jones, and Chung-chieh Shan. Fun with type functions. *Reflections on the Work of CAR Hoare*, pages 301–331, 2010. URL https://www.microsoft.com/en-us/research/wp-content/uploads/2016/07/typefun.pdf?from=https://research.microsoft.com/~simonpj/papers/assoc-types/fun-with-type-funs/typefun.pdf&type=exact.

Mark P Jones. Type classes with functional dependencies. In *European Symposium on Programming*, pages 230–244. Springer, 2000. URL https://doi.org/10.1007/3-540-46425-5_15.

Nada Amin, Tiark Rompf, and Martin Odersky. Foundations of path-dependent types. *Acm Sigplan Notices*, 49(10):233–249, 2014. URL http://lampwww.epfl.ch/~amin/dot/fpdt_post.pdf.

Jan Stolarek, Simon Peyton Jones, and Richard A Eisenberg. Injective type families for haskell. *ACM SIGPLAN Notices*, 50(12):118–128, 2015. URL https://repository.brynmawr.edu/cgi/viewcontent.cgi?article=1070&context=compsci_pubs.

Richard A Eisenberg and Jan Stolarek. Promoting functions to type families in haskell. *ACM SIGPLAN Notices*, 49(12):95–106, 2014. URL https://repository.brynmawr.edu/cgi/viewcontent.cgi?article=1000&context=compsci_pubs.

Csongor Kiss, Tony Field, Susan Eisenbach, and Simon Peyton Jones. Higher-order type-level programming in haskell. *Proceedings of the ACM on Programming Languages*, 3(ICFP):1–26, 2019. URL https://dl.acm.org/doi/pdf/10.1145/3341706.

Dominic Orchard and Tom Schrijvers. Haskell type constraints unleashed. In *International Symposium on Functional and Logic Programming*, pages 56–71. Springer, 2010. URL https://kar.kent.ac.uk/57498/1/constraint-families.pdf.

Edsko de Vries and Andres Löh. True sums of products. In *Proceedings of the 10th ACM SIGPLAN workshop on Generic programming*, pages 83–94, 2014. URL https://doi.org/10.1145/2633628.2633634.

Gert-Jan Bottu, Georgios Karachalias, Tom Schrijvers, Bruno C d S Oliveira, and Philip Wadler. Quantified class constraints. *ACM SIGPLAN Notices*, 52(10):148–161, 2017. URL https://www.pure.ed.ac.uk/ws/portalfiles/portal/42495988/quantcc.pdf.

Martin Sulzmann, Manuel MT Chakravarty, Simon Peyton Jones, and Kevin Donnelly. System f with type equality coercions. In *Proceedings of the 2007 ACM SIGPLAN international workshop on Types in languages design and implementation*, pages 53–66, 2007b. URL https://doi.org/10.1145/1190315.1190324.

Dimitrios Vytiniotis, Simon Peyton Jones, Tom Schrijvers, and Martin Sulzmann. Outsidein (x) modular type inference with local assumptions. *Journal of functional programming*, 21(4-5): 333–412, 2011. URL https://doi.org/10.1017/S0956796811000098.

Joachim Breitner, Richard A Eisenberg, Simon Peyton Jones, and Stephanie Weirich. Safe zero-cost coercions for haskell. In *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming*, pages 189–202, 2014. URL https://repository.brynmawr.edu/cgi/viewcontent.cgi?article=1013&context=compsci_pubs.

Simon Peyton Jones, Stephanie Weirich, Richard A Eisenberg, and Dimitrios Vytiniotis. A reflection on types. In *A List of Successes That Can Change the World: Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*, pages 292–317. Springer, 2016. URL https://repository.brynmawr.edu/cgi/viewcontent.cgi?article=1002&context=compsci_pubs.

Simon Marlow, Simon Peyton Jones, Andrew Moran, and John Reppy. Asynchronous exceptions in haskell. In *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, pages 274–285, 2001. URL https://classes.cs.uchicago.edu/archive/2007/spring/32102-1/papers/p274-marlow.pdf.

Simon Marlow. An extensible dynamically-typed hierarchy of exceptions. In *Proceedings of the 2006 ACM SIGPLAN Workshop on Haskell*, pages 96–106, 2006. URL https://archive.alvb.in/msc/03_infoafp/papers/2012-12-18_HoorCollege_MarlowExtensibleExceptions_dk.pdf.

Simon Peyton Jones. Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in haskell. *NATO SCIENCE SERIES SUB SERIES III COMPUTER*

AND SYSTEMS SCIENCES, 180:47–96, 2001. URL https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=2e6c9d76f9cb690dc18019fc894ba9572a8c2812.

Ralf Hinze, Jennifer Hackett, and Daniel WH James. Functional pearl: F for functor. 2012. URL www.cs.ox.ac.uk/people/daniel.james/functor/functor.pdf.

Ralf Hinze and Daniel WH James. Reason isomorphically! In *Proceedings of the 6th ACM SIGPLAN workshop on Generic programming*, pages 85–96, 2010. URL http://www.cs.ox.ac.uk/people/ralf.hinze/publications/WGP10.pdf.

Gérard Huet. The zipper. *Journal of functional programming*, 7(5):549–554, 1997. URL https://doi.org/10.1017/S0956796897002864.

Conor McBride. The derivative of a regular type is its type of one-hole contexts. *Unpublished manuscript*, pages 74–88, 2001. URL https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=7de4f6fddb11254d1fd5f8adfd67b6e0c9439eaa.

Michael Abbott, Thorsten Altenkirch, Neil Ghani, and Conor McBride. Derivatives of containers. In *International Conference on Typed Lambda Calculi and Applications*, pages 16–30. Springer, 2003. URL http://www.strictlypositive.org/derivcont.pdf.

Erik Meijer, Maarten Fokkinga, and Ross Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In *Conference on functional programming languages and computer architecture*, pages 124–144. Springer, 1991. URL https://ris.utwente.nl/ws/portalfiles/portal/6142049/meijer91functional.pdf.

Erik Meijer and Graham Hutton. Bananas in space: Extending fold and unfold to exponential types. In *Proceedings of the seventh international conference on Functional programming languages and computer architecture*, pages 324–333, 1995. URL https://dl.acm.org/doi/pdf/10.1145/224164.224225.

Olivier Danvy and Kevin Millikin. Refunctionalization at work. *Science of Computer Programming*, 74(8):534–549, 2009. ISSN 0167-6423. doi: https://doi.org/10.1016/j.scico.2007.10.007. URL https://www.sciencedirect.com/science/article/pii/S0167642309000227. Special Issue on Mathematics of Program Construction (MPC 2006).

Philip Wadler. Deforestation: Transforming programs to eliminate trees. In *European Symposium on Programming*, pages 344–358. Springer, 1988. URL https://link.springer.com/content/pdf/10.1007/3-540-19027-9_23.pdf.

СА Романенко. ВЫЯВЛЕНИЕ И ДОКАЗАТЕЛЬСТВО СВОЙСТВ ФУНКЦИОНАЛЬНЫХ ПРОГРАММ МЕТОДАМИ СУПЕРКОМПИЛЯЦИИ. URL https://www.keldysh.ru/council/1/klyuchnikov-diss.pdf.

Andrew Gill, John Launchbury, and Simon L Peyton Jones. A short cut to deforestation. In *Proceedings of the conference on Functional programming languages and computer architecture*, pages 223–232, 1993. URL https://dl.acm.org/doi/pdf/10.1145/165180.165214.

Jeremy Gibbons. Unfolding abstract datatypes. In *International Conference on Mathematics of Program Construction*, pages 110–133. Springer, 2008. URL https://ora.ox.ac.uk/objects/uuid:b51d2af6-25de-41ae-b3e7-9ac997de87b3/files/m7c19077c7bfba562a6cc9ceb3db62641.

William R Cook. On understanding data abstraction, revisited. In *Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pages 557–572, 2009. URL https://www.phaazon.net/media/uploads/on_understanding_data_abstraction.pdf.

Duncan Coutts, Roman Leshchinskiy, and Don Stewart. Stream fusion: From lists to streams to nothing at all. *ACM SIGPLAN Notices*, 42(9):315–326, 2007. URL https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=a4fad0d182605fcd155bebe3d620b7ffa0456968.

Oleg Kiselyov, Aggelos Biboudis, Nick Palladinos, and Yannis Smaragdakis. Stream fusion, to completeness. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 285–299, 2017. URL https://arxiv.org/pdf/1612.06668.

Paul Downen, Zachary Sullivan, Zena M Ariola, and Simon Peyton Jones. Codata in action. In *European Symposium on Programming*, pages 119–146. Springer International Publishing Cham, 2019. URL https://library.oapen.org/bitstream/handle/20.500.12657/23330/1006825.pdf?sequence=1#page=132.

Jon Louis Bentley. Little languages. *Commun. ACM*, 29(8):711–721, 1986. URL https://doi.org/10.1145/6424.315691.

Jeremy Gibbons and Nicolas Wu. Folding domain-specific languages: deep and shallow embeddings (functional pearl). In *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming*, pages 339–347, 2014. URL https://ora.ox.ac.uk/objects/uuid:0547e439-48d5-4749-a5eb-811e65cfaaee/files/mf31b32c6dcbd81dd6a8ef43166a8cc91.

Jeremy Gibbons. Functional programming for domain-specific languages. In *Central European Functional Programming School*, pages 1–28. Springer, 2013. URL https://ora.ox.ac.uk/objects/uuid:489909da-8818-4d6e-a4fd-0107b589ee76/files/m00be42b36cabbc6d11c8748fac8c7c71.

Benjamin C Pierce, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, and Brent Yorgey. Software foundations. *Webpage: http://www. cis. upenn. edu/bcpierce/sf/current/index. html*, 16, 2010. URL https://idris-hackers.github.io/software-foundations/pdf/sf-idris-2018.pdf.

Graham Hutton. Fold and unfold for program semantics. *ACM SIGPLAN Notices*, 34(1):280–288, 1998. URL https://dl.acm.org/doi/abs/10.1145/291251.289457.

Robert Nystrom. *Crafting Interpreters*. URL https://craftinginterpreters.com/contents.html.

Dom Hutchinson. Language engineering - notes. Course COMS22201 notes. URL https://github.com/dajhutchinson/Language-Engineering.

Simon Marlow. Parallel and concurrent programming in haskell. In *Central European Functional Programming School*, pages 339–401. Springer, 2011. URL https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=bd127ea1952996864c1542e1453973a78973ad5c.

Arthur Charguéraud. The locally nameless representation. *Journal of automated reasoning*, 49: 363–408, 2012. URL https://www.charguer aud.org/research/2009/ln/main.pdf.

Stephen Chong. *CS153: Compilers Lecture 12:Closures and Environments*. URL https://groups.seas.harvard.edu/courses/cs153/2018fa/lectures/Lec12-Functions.pdf.

Oleg Kiselyov. Typed tagless final interpreters. In *Generic and indexed programming: International spring school, sSGIP 2010, oxford, uK, march 22-26, 2010, revised lectures*, pages 130–174. Springer, 2012a. URL https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=f0b6a04331821a294bad4d08975637b4f363ec5a#page=136.

Frank Pfenning and Conal Elliott. Higher-order abstract syntax. *ACM sigplan notices*, 23(7): 199–208, 1988. URL https://doi.org/10.1145/960116.54010.

Jeff Epstein, Andrew P Black, and Simon Peyton-Jones. Towards haskell in the cloud. In *Proceedings of the 4th ACM symposium on Haskell*, pages 118–129, 2011. URL https://www.microsoft.com/en-us/research/wp-content/uploads/2016/07/remote.pdf?from=https://research.microsoft.com/en-us/um/people/simonpj/papers/parallel/remote.pdf&type=exact.

Jacques Carette, Oleg Kiselyov, and Chung-chieh Shan. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. In *Programming Languages and Systems: 5th Asian Symposium, APLAS 2007, Singapore, November 29-December 1, 2007. Proceedings 5*, pages 222–238. Springer, 2007. URL https://www.cambridge.org/core/journals/journal-of-functional-programming/article/finally-tagless-partially-evaluated-tagless-staged-interpreters-for-simpler-typed-lang7B2DC44A2127EBBA71ADE63809D9425F.

Eugenio Moggi. *Computational lambda-calculus and monads*. University of Edinburgh, Department of Computer Science, Laboratory for ..., 1988. URL http://www.lfcs.inf.ed.ac.uk/reports/88/ECS-LFCS-88-66/ECS-LFCS-88-66.pdf.

Philip Wadler. Comprehending monads. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, pages 61–78, 1990. URL https://dl.acm.org/doi/pdf/10.1145/91556.91592.

Philip Wadler. The essence of functional programming. In *Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–14, 1992. URL https://dl.acm.org/doi/pdf/10.1145/143165.143169.

Conor McBride and Ross Paterson. Applicative programming with effects. *Journal of functional programming*, 18(1):1–13, 2008. URL https://doi.org/10.1017%2FS0956796807006326.

John C Reynolds. The discoveries of continuations. *Lisp and symbolic computation*, 6:233–247, 1993. URL https://ics.uci.edu/~jajones/INF102-S18/readings/08_histcont.pdf.

Peter J Landin. Histories of discoveries of continuations: Belles-lettres with equivocal tenses. In *Proceedings of the Second ACM SIGPLAN Workshop on Continuations (CW'97), Technical report BRICS NS-96-13, University of Aarhus*, page 1. Citeseer, 1997. URL https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=14c609276166517ed1afb56e6b84a41ba157030b.

Steven E Ganz, Daniel P Friedman, and Mitchell Wand. Trampolined style. In *Proceedings of the fourth ACM SIGPLAN international conference on Functional programming*, pages 18–27, 1999. URL https://doi.org/10.1145/317636.317779.

RO Bjarnarson. Stackless scala with free monads. *Scala Days*, 2012. URL https://days2012.scala-lang.org/sites/days2012/files/bjarnason_trampolines.pdf.

Conor McBride. Clowns to the left of me, jokers to the right (pearl) dissecting data structures. In *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 287–295, 2008. URL https://web.archive.org/web/20170705104305id_/http://www.cis.upenn.edu/~cis39903/static/clowns.pdf.

Daniel Hillerström. Foundations for programming and implementing effect handlers. 2022. URL http://dx.doi.org/10.7488/era/2122.

Serkan Muhcu, Philipp Schuster, Michel Steuwer, and Jonathan Immanuel Brachthäuser. Multiple resumptions and local mutable state, directly. *Proceedings of the ACM on Programming Languages*, 9(ICFP):704–733, 2025. URL https://doi.org/10.1145/3747529.

Olivier Danvy. Defunctionalized interpreters for programming languages. *ACM Sigplan Notices*, 43 (9):131–142, 2008. URL http://jfla.inria.fr/2014/danvy-ICFP08.pdf.

Jeremy Gibbons. Continuation-passing style, defunctionalization, accumulations, and associativity. *arXiv preprint arXiv:2111.10413*, 2021. URL https://doi.org/10.48550/arXiv.2111.10413.

Olivier Danvy. An analytical approach to program as data objects, 2006. URL https://www.cs.tufts.edu/comp/150FP/archive/olivier-danvy/d_scientia.pdf.

Atze van der Ploeg and Oleg Kiselyov. Reflection without remorse: revealing a hidden sequence to speed up monadic reflection. In *Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell*, pages 133–144, 2014. URL https://doi.org/10.1145/2633357.2633360.

Pierre-Louis Curien and Hugo Herbelin. The duality of computation. *ACM sigplan notices*, 35(9): 233–243, 2000. URL https://doi.org/10.1145/357766.351262.

R Kent Dyvbig, Simon Peyton Jones, and Amr Sabry. A monadic framework for delimited continuations. *Journal of functional programming*, 17(6):687–730, 2007. URL https://doi.org/10.1017/S0956796807006259.

Andrzej Filinski. Representing monads. In *Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 446–457, 1994. URL https://doi.org/10.1145/174675.178047.

Ana Lúcia De Moura and Roberto Ierusalimschy. Revisiting coroutines. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 31(2):1–31, 2009. URL https://doi.org/10.1145/1462166.1462167.

Roman Elizarov, Mikhail Belyaev, Marat Akhin, and Ilmir Usmanov. Kotlin coroutines: design and implementation. In *Proceedings of the 2021 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, pages 68–84, 2021. URL https://www.researchgate.net/profile/Mikhail-Belyaev-3/publication/355361443_Kotlin_coroutines_design_and_implementation/links/624c63c34f88c3119ce12a49/Kotlin-coroutines-design-and-implementation.pdf.

Anders Møller and Michael I Schwartzbach. Static program analysis. *Notes. Feb*, 2012. URL https://users-cs.au.dk/amoeller/spa/spa.pdf.

Yannick Forster, Ohad Kammar, Sam Lindley, and Matija Pretnar. On the expressive power of user-defined effects: Effect handlers, monadic reflection, delimited control. *Proceedings of the ACM on Programming Languages*, 1(ICFP):1–29, 2017. URL https://doi.org/10.1145/3110257.

Ningning Xie, Jonathan Immanuel Brachthäuser, Daniel Hillerström, Philipp Schuster, and Daan Leijen. Effect handlers, evidently. *Proceedings of the ACM on Programming Languages*, 4 (ICFP):1–29, 2020. URL https://doi.org/10.1145/3408981.

Oleg Kiselyov, Amr Sabry, and Cameron Swords. Extensible effects: an alternative to monad transformers. *ACM SIGPLAN Notices*, 48(12):59–70, 2013. URL https://doi.org/10.1145/2578854.2503791.

Andrej Bauer. What is algebraic about algebraic effects and handlers? *arXiv preprint arXiv:1807.05923*, 2018. URL https://doi.org/10.48550/arXiv.1807.05923.

Daniel Hillerström and Sam Lindley. Shallow effect handlers. In *Asian Symposium on Programming Languages and Systems*, pages 415–435. Springer, 2018. URL https://www.research.ed.ac.uk/files/76099718/shallow_effect_handlers.pdf.

Wouter Swierstra. Data types à la carte. *Journal of functional programming*, 18(4):423–436, 2008. URL https://www.cs.tufts.edu/~nr/cs257/archive/wouter-swierstra/DataTypesALaCarte.pdf.

Oleg Kiselyov and Hiromi Ishii. Freer monads, more extensible effects. *ACM SIGPLAN Notices*, 50 (12):94–105, 2015. URL https://doi.org/10.1145/2887747.2804319.

Oleg Kiselyov. Iteratees. In *FLOPS'12: Proceedings of the 11th international conference on Functional and Logic Programming*, 2012b. URL https://okmij.org/ftp/Haskell/Iteratee/describe.pdf.

Daan Leijen. Algebraic effect handlers with resources and deep finalization. Technical report, Tech. Rep. MSR-TR-2018-10, Microsoft Research (April 2018), 2018. URL https://www.microsoft.com/en-us/research/wp-content/uploads/2018/04/resource-v1.pdf.

KC Sivaramakrishnan, Stephen Dolan, Leo White, Tom Kelly, Sadiq Jaffer, and Anil Madhavapeddy. Retrofitting effect handlers onto ocaml. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 206–221, 2021. URL https://doi.org/10.1145/3453483.3454039.

Luna Phipps-Costin, Andreas Rossberg, Arjun Guha, Daan Leijen, Daniel Hillerström, KC Sivaramakrishnan, Matija Pretnar, and Sam Lindley. Continuing webassembly with effect handlers. *Proceedings of the ACM on Programming Languages*, 7(OOPSLA2):460–485, 2023. URL https://doi.org/10.1145/3622814.

Nicolas Wu and Tom Schrijvers. Fusion for free: Efficient algebraic effect handlers. In *International Conference on Mathematics of Program Construction*, pages 302–322. Springer, 2015. URL https://lirias.kuleuven.be/retrieve/322544.

Sheng Liang, Paul Hudak, and Mark Jones. Monad transformers and modular interpreters. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 333–343, 1995. URL https://dl.acm.org/doi/pdf/10.1145/199448.199528.

Mark P Jones. Functional programming with overloading and higher-order polymorphism. In *International School on Advanced Functional Programming*, pages 97–136. Springer, 1995. URL http://web.cecs.pdx.edu/mpj/pubs/springschool95.pdf.

Tom Schrijvers, Maciej Piróg, Nicolas Wu, and Mauro Jaskelioff. Monad transformers and modular algebraic effects: what binds them together. In *Proceedings of the 12th ACM SIGPLAN International Symposium on Haskell*, pages 98–113, 2019. URL https://doi.org/10.1145/3331545.3342595.

Gordon Plotkin and John Power. Notions of computation determine monads. In *International Conference on Foundations of Software Science and Computation Structures*, pages 342–356. Springer, 2002. URL https://era.ed.ac.uk/bitstream/handle/1842/196/Comp_Eff_Monads.pdf?sequence=1.

Gordon D Plotkin and Matija Pretnar. Handling algebraic effects. *Logical methods in computer science*, 9, 2013. URL https://doi.org/10.2168/LMCS-9(4:23)2013.

Nicolas Wu, Tom Schrijvers, and Ralf Hinze. Effect handlers in scope. In *Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell*, pages 1–12, 2014. URL https://doi.org/10.1145/2633357.2633358.

Zhixuan Yang, Marco Paviotti, Nicolas Wu, Birthe van den Berg, and Tom Schrijvers. Structured handling of scoped effects. In *European Symposium on Programming*, pages 462–491. Springer International Publishing Cham, 2022. URL https://library.oapen.org/bitstream/handle/20.500.12657/54028/1/978-3-030-99336-8.pdf#page=476.

Cas van der Rest, Jaro Reinders, and Casper Bach Poulsen. Handling higher-order effects. *arXiv preprint arXiv:2203.03288*, 2022. URL https://arxiv.org/pdf/2203.03288.

Yizhou Zhang, Guido Salvaneschi, and Andrew C Myers. Handling bidirectional control flow. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–30, 2020. URL https://doi.org/10.1145/3428207.