Оглавление

Предисловие от издательства

Отзывы и пожелания

Мы всегда рады отзывам наших читателей. Расскажите нам, что вы думаете об этой книге – что понравилось или, может быть, не понравилось. Отзывы важны для нас, чтобы выпускать книги, которые будут для вас максимально полезны.

Вы можете написать отзыв на нашем сайте www.dmkpress.com, зайдя на страницу книги и оставив комментарий в разделе «Отзывы и рецензии». Также можно послать письмо главному редактору по адресу dmkpress@gmail.com; при этом укажите название книги в теме письма.

Если вы являетесь экспертом в какой-либо области и заинтересованы в написании новой книги, заполните форму на нашем сайте по адресу http://dmkpress.com/authors/publish_book/ или напишите в издательство по адресу dmkpress@gmail.com.

Список опечаток

Хотя мы приняли все возможные меры для того, чтобы обеспечить высокое качество наших текстов, ошибки все равно случаются. Если вы найдете ошибку в одной из наших книг – возможно, ошибку в основном тексте или программном коде, – мы будем очень благодарны, если вы сообщите нам о ней. Сделав это, вы избавите других читателей от недопонимания и поможете нам улучшить последующие издания этой книги.

Если вы найдете какие-либо ошибки в коде, пожалуйста, сообщите о них главному редактору по адресу dmkpress@gmail.com, и мы исправим это в следующих тиражах.

Нарушение авторских прав

Пиратство в интернете по-прежнему остается насущной проблемой. Издательство «ДМК Пресс» очень серьезно относится к вопросам защиты авторских прав и лицензирования. Если вы столкнетесь в интернете с незаконной публикацией какой-либо из наших книг, пожалуйста, пришлите нам ссылку на интернет-ресурс, чтобы мы могли применить санкции.

Ссылку на подозрительные материалы можно прислать по адресу dmkpress@gmail.com.

Мы высоко ценим любую помощь по защите наших авторов, благодаря которой мы можем предоставлять вам качественные материалы.

Вступительное слово автора

Дорогие друзья!

Спасибо за интерес, проявленный к книге «Проектирование на уровне типов: системный взгляд на дизайн и архитектуру»! Эта книга станет вашим гидом по очаровательному миру программирования на типах и сделает его таким же доступным, как легкая прогулка по лесу. Основное повествование книги ведется на языке Haskell, но также в книге приведен раздел под названием «Розеттский камень», где представлено переложение тех же идей и подходов на Rust и Scala 3.

Вы, возможно, знаете, что я написал также книгу «Functional Design and Architecture: Examples in Haskell» (Manning Publications, 2024, далее FDaA), глубокое и новаторское исследование практического программирования приложений на функциональных языках. В ней рассмотрены общие идеи, принципы проектирования и лучшие практики разработки надежных приложений, что делает ее всесторонним источником знаний о программной инженерии. Я ставил себе целью снабдить читателя всем необходимым для эффективного проектирования и реализации приложений с помощью функциональных подходов.

Однако в процессе работы над FDaA мне стало ясно, что миру типов недостает как хороших практик, так и хороших обучающих материалов. Я решил заполнить и этот пробел и попутно сформулировать подход к предмету, отличный от принятого в других источниках.

Программирование на уровне типов существует уже довольно давно, и многие языки предлагают развитые системы типов. Разработчики любят типы, особенно в Haskell (а также Rust и Scala 3), где это одно из самых притягательных языковых средств. Пишущие на Haskell часто тяготеют к математическим основаниям типов, в частности к теории категорий и теории типов. Однако, пытаясь углубить свое понимание типов, я нередко сдавался – раздосадованный и вконец запутавшийся. Академический акцент на математику не давал тех ответов, которые я искал.

Я прагматик, поэтому мне нужна была ясная и практическая методология проектирования ПО с помощью типов, которая охватывала бы все, начиная с принципов проектирования и заканчивая деталями реализации. Но почти все источники говорили о другом. Они были либо чрезмерно академическими, либо настолько фрагментированными, что общая картина никак не складывалась. Программирование на уровне типов – непростая тема, но в Haskell это проявляется с особенной силой из-за разрозненности и сложности материалов. Зачастую источники больше напоминают плохо написанный фольклор или крупицы мудрости, чем структурированное, практически ориентированное знание.

Понимать предмет и хорошо его преподавать не одно и то же. Особенно если отсутствует прагматика, а присутствует чистое эстетство. Если вы испытывае-

те схожие чувства, то эта книга для вас. Она написана в неформальном занимательном стиле, изобилует практическими примерами и свободна от ненужной математической сложности. Полученные знания можно будет сразу же применить для реальных задач.

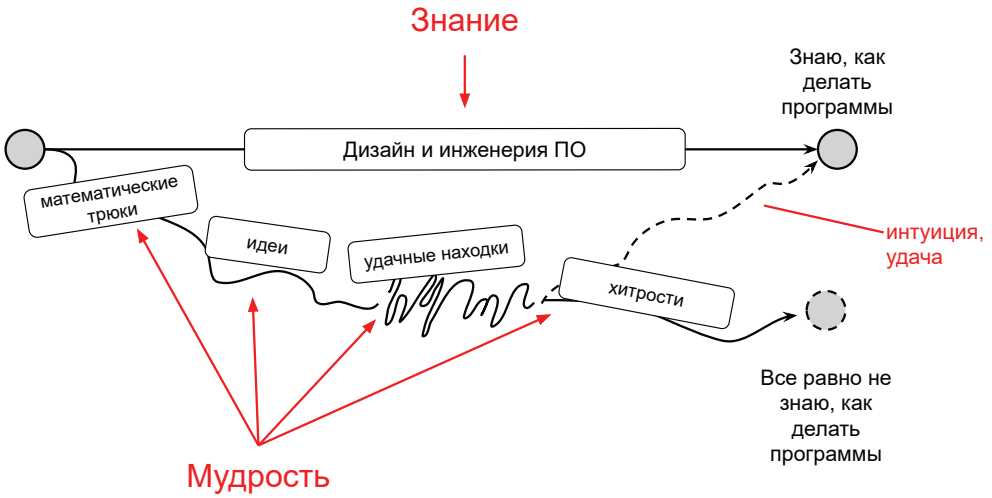

Рис. Мудрость и знание

В своей книге я превращаю программирование на типах в практичную и хорошо организованную дисциплину, свободную от излишних академических абстракций и сфокусированную на реальных приложениях. В сочетании с *функциональным декларативным проектированием* (Functional Declarative Design – FDD) – методологией, которую я развиваю в книге FDaA, – мы теперь имеем методологию *прагматичного проектирования на уровне типов* (Pragmatic Type-Level Design – PTLD).

Ссылки

Alexander Granin «Functional Design and Architecture: Examples in Haskell» (Manning Publications, 2024)
https://www.manning.com/books/functional-design-and-architecture

Код и дополнительные материалы к данной книге:
https://github.com/graninas/Pragmatic-Type-Level-Design

Английская версия Pragmatic Type-Level Design, опубликованная на LeanPub:
https://leanpub.com/pragmatic-type-level-design

Буду рад, если книга вам понравится.

С наилучшими пожеланиями,
Александр Гранин

ЧАСТЬ I

Основы проектирования программ на уровне типов

Глава 1

Эмерджентный дизайн на уровне типов

Краткое содержание этой главы

- Как программировать на типах и на значениях.
- Что такое прагматичное проектирование на уровне типов.
- Основы программирования на типах.

Что такое проектирование на уровне типов? Прежде чем ответить на этот вопрос, проведу неожиданную параллель между программированием на типах и клеточными автоматами. Это не просто сравнение по отдаленной ассоциации – философская связь между тем и другим гораздо глубже. Мы убедимся в этом, когда будем работать над приложением клеточных автоматов на уровне типов.

Любите ли вы клеточные автоматы, как люблю их я? Какое же это замечательное изобретение – великолепная игра «Жизнь» Джона Конвея! Впечатляющая глубина игры «Жизнь» проистекает из очень простого набора правил, который поражает воображение при первом знакомстве с этим клеточным автоматом. Всего два правила, руководящих эволюцией клеточного мира, – и вы получаете сложнейшую вселенную, полную странных тварей, которые двигаются, осциллируют, сталкиваются, сливаются, расходятся, рождаются и умирают.

Игра «Жизнь» полна по Тьюрингу, поэтому не будет ошибкой назвать ее эзотерическим двумерным графическим языком программирования. Кодирование на этом языке обладает особой прелестью. На бесконечной квадратной сетке мы включаем и выключаем клетки, рисуя тем самым программу, которая должна сделать что-то полезное. Правила, таким образом, играют роль компьютера, который распознает эту монохромную программу и шаг за шагом преобразует входной мир в выходной. Мы готовим начальную конфигурацию,

исполняем мир и после определенного числа итераций получаем результат, надеясь, что он обладает какими-то желательными свойствами.

Итеративная эволюция этого мира детерминирована. Как и в чистом функциональном программировании, начав с одинаковой конфигурации клеток, мы получаем одинаковые результаты, но, в отличие от программирования, предсказать результат, зная начальную конфигурацию, гораздо труднее, даже для небольшого числа итераций. Программы в игре «Жизнь» крайне восприимчивы к дефектам. Малейшая ошибка, неправильно включенная клетка – и ваш тщательно устроенный мир переходит в совершенно неожиданное состояние. То, что когда-то было узнаваемой картиной, превращается в хаос, наполненный танцующими клеточными демонами. То, что было осмысленной информацией, искажается и вырождается в простые примитивы, а иногда в полное ничто.

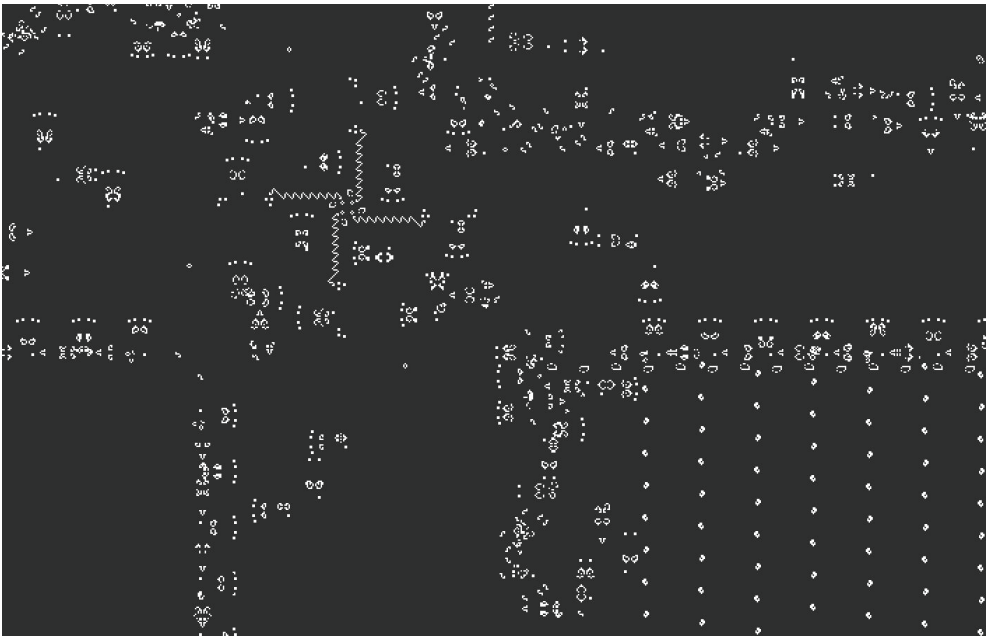

Рис. 1.1. Metapixel: игра «Жизнь», встроенная в игру «Жизнь» (фрагмент)

Именно поэтому программирование такой системы – весьма трудная задача. Не помогает даже знание правил игры «Жизнь», как и то, что она полна по Тьюрингу. Мы программируем «Жизнь» методом проб и ошибок, хотя и выработано немало средств, призванных облегчить этот процесс. Клеточные паттерны проектирования, готовые решения, конфигурируемые библиотеки механизмов, техники отладки – все это помогает справиться со сложностью, естественно вытекающей из простых базовых предпосылок.

Звучит знакомо, правда? Действительно, все сказанное применимо и к программированию на типах. Свое начало оба берут в теоретической информатике. Игра «Жизнь» и проектирование на уровне типов обещают массу удовольствия от экспериментов. Как и клеточная программа, программа

на типах начинается с простых идей, которые быстро разрастаются в сложнейшее сооружение, за которым бывает непросто уследить. Часто эволюцию таких программ трудно или невозможно предсказать. Как и в игре «Жизнь», крохотное неверное изменение кода на уровне типов может привести к нежелательным последствиям.

Будучи обоюдоострым орудием, программирование на типах может серьезно поранить, и это обязательно произойдет, если обращаться с ним неумело. Ввиду сложности в истории «Жизни» появились разные практики, и почему бы им не появиться в программировании на типах. Писать большие, но при этом управляемые программы с широким использованием типов следует осторожно, применяя разумные и контролируемые подходы. Иными словами, программирование на типах должно стать инженерной дисциплиной, а не просто искусством или набором хитроумных приемов. Благодаря прагматичности высказываемых в книге идей многие программные миры избегут мучительной и дорогостоящей смерти от тысячи порезов.

1.1. На пути к проектированию на уровне типов

Все эмерджентные системы одинаковы. Располагая небольшим набором простых кусочков и правилами их соединения, мы можем построить по-настоящему сложный механизм, служащий каким-то целям.

Это верно для биологической жизни, основанной на ДНК. Любая двойная спираль образована четырьмя нуклеотидами, но количество форм жизни, порождаемых таким кодированием, потрясает воображение.

С игрой «Жизнь» то же самое. Правило включения-выключения клеток на двумерной сетке поймет любой ребенок, но оно порождает миры, которые мы прежде не видывали.

Таково и проектирование на уровне типов.

> **ОПРЕДЕЛЕНИЕ.** ***Проектирование на уровне типов (type-level design)***: проектирование программного обеспечения с упором на применение различных средств и приемов на уровне типов для решения обычных проблем программирования на этапе компиляции. При проектировании на уровне типов предпочтение отдается кодированию в типах, а не в значениях. Обобщенный код становится предпочтительнее конкретного, а вычисления на этапе компиляции подменяют вычисления на этапе выполнения.

> **ОПРЕДЕЛЕНИЕ.** ***Средство уровня типов (type-level feature):*** любое языковое средство, имеющее отношение к типам и преобразованиям типов.

> **ОПРЕДЕЛЕНИЕ.** ***Проектирование на уровне значений (value-level design).*** Задачи программирования и бизнес-логика выражаются с помощью значений и функций. Типы в основном описывают данные и редко служат иным целям. Проектирование на уровне значений можно назвать еще программированием, ориентированным на данные.

ПОДСКАЗКА. Реальные проекты на статически типизированных языках всегда являются смесью решений на уровне типов и на уровне значений.

Системы типов в таких языках, как Haskell и Scala, хорошо развиты и наполнены сложными идеями, но даже подмножество, состоящее из базовых инструментов, способно помочь в повседневной практике. Вы удивитесь, сколь многого можно достичь, не хватаясь за каждую блестящую погремушку в системе типов. Меньший набор средств, хорошо подходящих друг к другу, для нас ценнее, чем большой набор плохо сочетающихся средств, которые могут превратить код в хаос. Таким образом, моя методология проектирования на уровне типов тоже стремится быть эмерджентной системой: из нескольких базовых идей строится мощный и внутренне богатый инструментарий.

Прежде чем пускаться во все тяжкие, давайте обозначим базовые понятия и обсудим, как выглядит программирование с применением типов и значений.

1.1.1. Типы и значения

Если бы нас попросили написать программу с примитивной функциональностью, мы, будучи инженерами, вряд ли стали бы выходить за рамки простых решений. Хорошим примером может послужить приложение для игры «Жизнь». Сформулируем требования:

- игра «Жизнь» является консольным приложением;
- приложение принимает количество итераций эволюции мира;
- приложение принимает входной файл, содержащий начальную конфигурацию мира в текстовом формате размером не более 100 Кб;
- приложение принимает имя выходного файла для сохранения конечной конфигурации мира в текстовом формате;
- приложение выполняет вычисления мира и выводит результаты в выходной файл.

Здесь перечислены самые важные функциональные и нефункциональные требования. В них ничего не говорится о производительности, и не требуется отслеживать промежуточные миры в процессе эволюции. Мы можем просто хранить текущий мир в памяти в структуре данных, похожей на массив или на словарь. Подобный способ хранения далек от оптимального, учитывая, что миры часто растут неограниченно, но мы хотя бы сможем быстро создать наше приложение.

В результате традиционного моделирования предметной области мы получаем обычный код, опирающийся на какие-то типы данных. Ничего хитроумного или обобщенного; просто типы, тегирующие значения, которыми должна оперировать наша программа. Следующий тип `Board` отображает индексы на состояния клеток (см. рис. 1.2):

```
type Coords = (Int, Int) #A
data Cell = Alive | Dead #B
type Board = Map Coords Cell #C

#A Координаты
#B Возможные состояния клеток
#C Словарь клеток (двумерное поле)
```

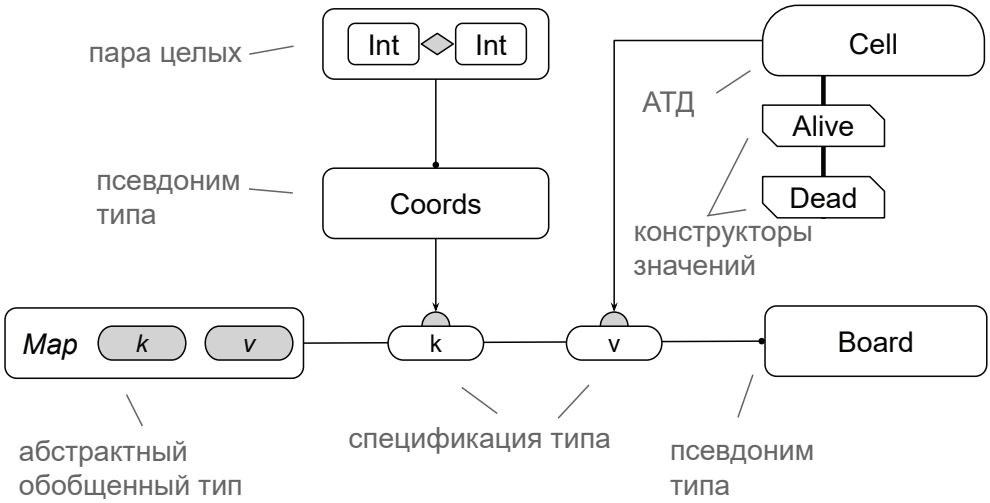

Рис. 1.2. Диаграмма типизированных форм для Board

> **ПОДСКАЗКА.** Полное руководство по диаграммам типизированных форм см. в приложении А.

Что может быть проще? Всего-то двумерный ключ, указывающий на конкретную клетку доски. Каждая клетка кодируется прямо, но перечислять мертвые клетки нет необходимости. Следующая фигура – планер – содержит пять живых клеток:

```
glider :: Board
glider = Map.fromList [((1, 0), Alive),
 ((2, 1), Alive),
 ((0, 2), Alive),
 ((1, 2), Alive),
 ((2, 2), Alive)]
```

Форма этого планера представлена на рис. 1.3.

```
  0 1 2
 +-------
0 | . x .
1 | . . x
2 | x x x
```

Рис. 1.3. Планер

Первоначальный код приложения может выглядеть как функция `main`, в которой сосредоточены все операции чтения-записи файлов и кое-какие зашитые константы. В коде ниже производится пять итераций эволюции мира:

```
main = do
 board1 <- loadFromFile "./data/glider.txt" #A
 let board2 = iterateWorld 5 board1 #B
 saveToFile "./data/glider_5th_gen.txt" board2 #C

loadFromFile :: FilePath -> IO Board #D
saveToFile :: FilePath -> Board -> IO ()
iterateWorld :: Int -> Board -> Board

#A Загрузить начальную конфигурацию мира
#B Выполнить пять шагов
#C Сохранить конечную конфигурацию мира
#D Функции, выполняющие конкретные операции
```

Типы здесь играют описательную роль. Функция `loadFromFile` возвращает значение типа `Board`, затем `iterateWorld` использует это значение для получения следующих пяти. Сам тип `Board` не делает ничего, разве что способствует удобочитаемости и понятности кода. Мы могли бы ограничиться типом `Map` и получить ту же функциональность, потому что эти типы – просто синонимы:

```
loadFromFile :: FilePath -> IO Board
loadFromFile :: FilePath -> IO (Map Coords Cell)

iterateWorld :: Int -> Board -> Board
iterateWorld :: Int -> Map Coords Cell -> Map Coords Cell
```

Ничего не изменилось, поэтому мы можем заключить, что идентификатор типа `Board` не играет существенной роли в нашем коде. Его присутствие объясняется удобством, но никакого дополнительного смысла он не несет.

Новый смысл можно было бы ввести, обернув этот синоним типа ключевым словом newtype (диаграмма типизированных форм показана на рис. 1.4):

```
newtype GoL = GoL Board
iterateGoL :: Int -> GoL -> GoL
```

ПОДСКАЗКА. В других языках есть средства, похожие на `newtype` из Haskell. Дополнительную информацию см. в части IV «Розеттский камень».

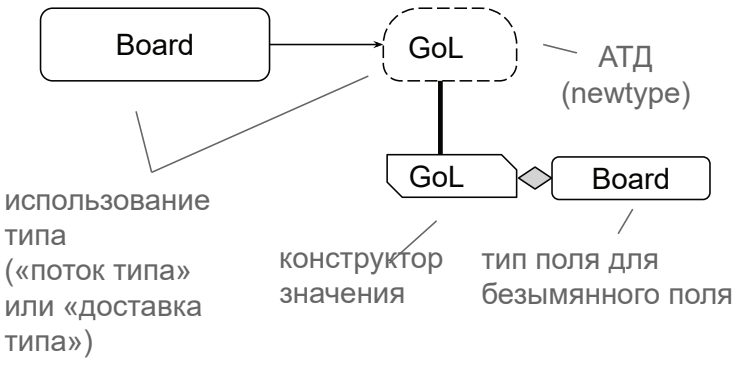

Рис. 1.4. Диаграмма типизированных форм для игры «Жизнь»