

Оглавление

Предисловие от издательства.....	10
Вступительное слово автора	11
ЧАСТЬ I. ОСНОВЫ ПРОЕКТИРОВАНИЯ ПРОГРАММ НА УРОВНЕ ТИПОВ.....	13
Глава 1. Эмерджентный дизайн на уровне типов.....	15
1.1. На пути к проектированию на уровне типов	17
1.1.1. Типы и значения	18
1.1.2. Прагматичное проектирование на уровне типов.....	22
1.1.3. Диаграммы типизированных форм	26
1.2. В предвкушении проектирования на уровне типов	26
1.2.1. Основы обобщенных типов.....	27
1.2.2. Основы классов типов	29
1.2.3. Принципы проектирования	32
1.2.4. Основы литералов уровня типа	33
1.3. Резюме	38
Глава 2. Вариант использования: простая расширяемость	39
2.1. Расширяемость.....	40
2.1.1. Механизмы расширяемости.....	40
2.1.2. Требование расширяемости.....	41
2.1.3. Точки расширения	45
2.2. Простое расширяемое приложение	47
2.2.1. Интерфейс класса типов	47
2.2.2. Проблема неоднородного хранилища.....	50
2.2.3. Функционализация и экзистентификация	53
2.2.4. Функционализированное хранилище	53
2.2.5. Экзистенциальное хранилище.....	55
2.3. Резюме	58
Глава 3. Вариант использования: универсальность и кастомизация	60
3.1. Универсальность на уровне типов	62
3.1.1. Пустые ADT	62
3.1.2. Применения типов.....	64
3.1.3. Обыкновенные и специальные роды	67
3.1.4. Пользовательские типы-ADT и роды.....	70
3.1.5. Типы-списки	73
3.2. Кастомизация на уровне типов.....	75
3.2.1. Методология проектирования через сценарии	76

3.2.2. Кастомизируемый eDSL уровня типов	78
3.3. Резюме	84
Глава 4. Вариант использования: обеспечение корректности.....	85
4.1. Корректность – это о смысле.....	86
4.1.1. Типобезопасность и корректность	86
4.1.2. Программирование на уровне типов и корректность.....	88
4.2. Статическая целостность.....	89
4.2.1. Усиление модели предметной области	90
4.2.2. Статические и изменчивые понятия предметной области.....	92
4.2.3. Статическая операционная целостность	95
4.2.4. Валидаторы на уровне типов	97
4.2.5. Статическая структурная целостность	100
4.3. Резюме	103
ЧАСТЬ II. ВОПРОСЫ АРХИТЕКТУРЫ ПРИЛОЖЕНИЙ НА УРОВНЕ ТИПОВ	105
Глава 5. Архитектура приложения	107
5.1. Подходы к архитектуре ПО	107
5.1.1. Архитектурные слои	108
5.1.2. Практика двукратного использования.....	110
5.1.3. Два приложения, одна архитектура.....	111
5.2. Структура приложения	113
5.2.1. Слоистая архитектура.....	113
5.2.2. Структура проекта	116
5.2.3. Организация кода уровня типов.....	117
5.2.4. Слой приложения.....	120
5.3. Резюме	122
Глава 6. Проектирование компонентов	123
6.1. Статические и динамические модели предметной области.....	124
6.1.1. Раздельные модели на уровне типов и на уровне значений	125
6.1.2. Паттерн проектирования Granular Type Selector.....	126
6.1.3. Интерпретация статических и динамических моделей.....	129
6.1.4. Статическая материализация	130
6.1.5. Объекты переноса данных и сериализация	133
6.2. Два функциональных интерфейса.....	134
6.2.1. Свойства истинного интерфейсного механизма.....	135
6.2.2. Сравнение класса типов и свободной монады	135
6.2.3. Интерфейс на базе свободной монады	137
6.2.4. Интерфейс на базе класса типов и паттерн Dynamic Payload.....	139
6.3. Резюме	142
ЧАСТЬ III. ПРОДВИНУТОЕ ПРОЕКТИРОВАНИЕ НА УРОВНЕ ТИПОВ	143

Глава 7. Вариант использования: ООП на уровне типов.....	145
7.1. Мультипарадигмальный подход	146
7.2. Беглый взгляд на ООП уровня типов	147
7.2.1. Zeplog: концепция	147
7.2.2. The Expression Problem.....	149
7.3. Типо-ориентированная объектная модель	151
7.3.1. Модель свойств	151
7.3.2. Статическая материализация и динамическое инстанцирование ...	155
7.3.3. Скрипты	158
7.3.4. Паттерн проектирования Typed-Untyped.....	160
7.4. Резюме.....	162
Глава 8. Вариант использования:	
продвинутая расширяемость.....	164
8.1. Продвинутое проектирование на уровне типов и расширяемость.....	166
8.1.1. Моделирование предметной области с помощью пустых параметризованных ADT	166
8.1.2. Интерфейсы уровня типов	167
8.1.3. Универсальный механизм вычислений	173
8.1.4. Продвинутая расширяемость и The Expression Problem.....	174
8.1.5. Комбинаторные eDSL уровня типов и лямбды	178
8.2. Резюме	181
Заключение	183
ЧАСТЬ IV. РОЗЕТТСКИЙ КАМЕНЬ.....	185
Розеттский камень, глава 1. Rust	188
RSC.1.1. Средства уровня значений и смешанного уровня	189
RSC.1.1.1. Newtype	189
RSC.1.1.2. ADT	190
RSC.1.1.3. Характеристики, интерфейсы и классы типов	191
RSC.1.1.4. Пустые ADT	192
RSC.1.1.5. Прокси-типы и фантомные типы.....	193
RSC.1.1.6. Простые обобщенные типы	195
RSC.1.1.7. Параметризованные ADT	195
RSC.1.2. Средства уровня типов.....	196
RSC.1.2.1. Типы-литералы.....	196
RSC.1.2.2. Пользовательские роды и ADT уровня типов	197
RSC.1.2.3. Типы-списки	200
RSC.1.2.4. Типы строки	201
RSC.1.2.5. Семейства типов и ассоциированные типы	203
RSC.1.3. Интерфейсы уровня типов, расширяемость и моделирование предметной области.....	204
RSC.1.3.1. Продвинутая система родов	206
RSC.1.3.2. Экзистенциальные обертки и типы реализации.....	206
RSC.1.3.3. Механизм вычислений.....	208
RSC.1.3.4. Типы-списки с родом	210

RSC.1.4. Контроль целостности.....	211
RSC.1.4.1. Равенство типов.....	211
RSC.1.4.2. Механизм выбора типа	212
RSC.1.4.3. Статические и изменчивые понятия предметной области....	213
RSC.1.4.4. Валидаторы целостности	214
RSC.1.4.5. Статическое утверждение	216
Розеттский камень, глава 2. Scala 3.....	218
RSC.2.1. Средства уровня значений и смешанного уровня	219
RSC.1.1.1. Newtype (непрозрачные типы)	219
RSC.2.1.2. ADT	221
RSC.2.1.3. Характеристики и классы типов.....	222
RSC.2.1.4. Пустые ADT, параметризованные ADT и прокси	223
RSC.2.2. Средства уровня типов.....	225
RSC.2.2.1. Типы-литералы.....	225
RSC.2.2.2. Пользовательские роды и ADT уровня типов	227
RSC.2.2.3. Типы-списки	228
RSC.2.2.4. Семейства типов и match-типы.....	230
RSC.2.3. Интерфейсы уровня типов, расширяемость и моделирование предметной области.....	231
RSC.2.3.1. Интерфейсы уровня типов и экзистенциальные обертки....	232
RSC.2.3.2. Типы-списки с родом	234
RSC.2.3.3. Механизм вычислений.....	235
RSC.2.4. Контроль целостности.....	237
RSC.2.4.1. Валидаторы целостности и сравнение типов.....	238
RSC.2.4.2. Автономные и интегрированные валидаторы.....	239
Приложение А. Диаграммы типизированных форм.....	243
A.1. Общие соглашения	244
Фигуры с закругленными и прямыми углами.....	244
Фигуры со сплошными и штриховыми границами	244
Стрелки и соединители	245
Факультативные элементы	245
Особые фигуры и фигуры, относящиеся к неизвестным типам	245
Комментарии	246
A.2. Обычные типы, пары, псевдонимы, списки.....	246
Обычные типы	246
Пары	246
Псевдонимы.....	246
Списки	247
A.3. Простые ADT	247
Новые типы.....	249
A.4. Параметризованные типы.....	249
Обычные параметризованные типы	249
Простой синтаксис для параметризованных псевдонимов.....	250
Абстрактные обобщенные типы	250
A.5. Синтаксис спецификации обобщенных типов	251
A.6. Классы типов и экземпляры	251

A.7. Роды	252
Обыкновенные типы	252
Конкретные роды	252
Конструкторы типов.....	252
A.8. Повышение типа.....	253
A.9. Семейства типов.....	254
A.10. Шаблон HKD	254
Приложение В. Экзистенциальный бойцовский клуб	256
В.1. Haskell, экзистентификация	257
В.2. Haskell, функционализация	258
В.3. Rust, экзистентификация.....	259
В.4. Rust, функционализация.....	260
В.5. C++, объектно ориентированный вариант.....	261
В.6. C++, функционализация.....	263
В.7. Scala 2, экзистентификация с имплицитами	265
Приложение С. Мифологизированная корректность.....	267
С.1. Типобезопасность.....	268
С.1.1. Обобщенная типобезопасность.....	268
С.1.2. Дескриптивная типобезопасность	270
С.2. Техническая корректность.....	271
С.2.1. Корректность структур данных и алгоритмов.....	272
С.2.2. Корректность моделей данных.....	274
С.2.3. Корректность языков.....	276
С.3. Заключение	280
Приложение D. Расширяемая модель значений в проекте Zeplog	281
D.1. Трихотомия: расширяемость, типобезопасность и простота	281
D.2. Расширяемая модель значений в Zeplog.....	283
D.2.1. Свойства и скрипты.....	284
D.2.2. Переменные, значения и теги	285
D.2.3. Расширяемость с помощью открытых семейств типов.....	288
Приложение E. Событийная архитектура игры Minefield	290
E.1. Архитектура приложения Minefield.....	290
E.1.1. Событийная модель акторов.....	291
E.1.2. Модель предметной области уровня типов для игры Minefield.....	293
E.1.3. Расширяемость существительных и глаголов предметной области	295
E.2. Реализация приложения Minefield	296
E.2.1. Паттерн MVar Request-Response	297
E.2.2. События и очереди	298
E.2.3. Реализация модели акторов.....	301
Предметный указатель	305

Вступительное слово автора

Дорогой друг!

Спасибо за интерес, проявленный к книге «Проектирование на уровне типов : системный взгляд на дизайн и архитектуру»! Эта книга станет твоим 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 крупницы 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 испытываете

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.

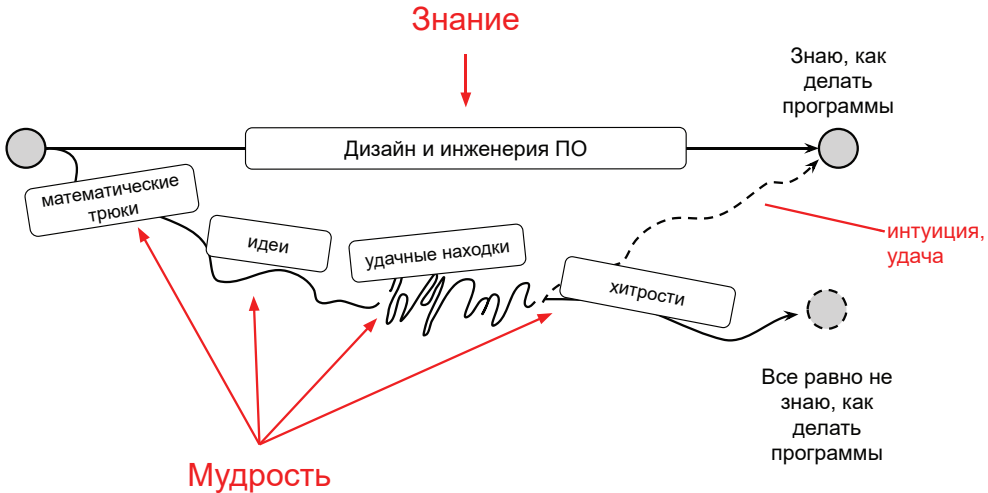


Рисунок. Мудрость и знание

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

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 избегают 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 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)

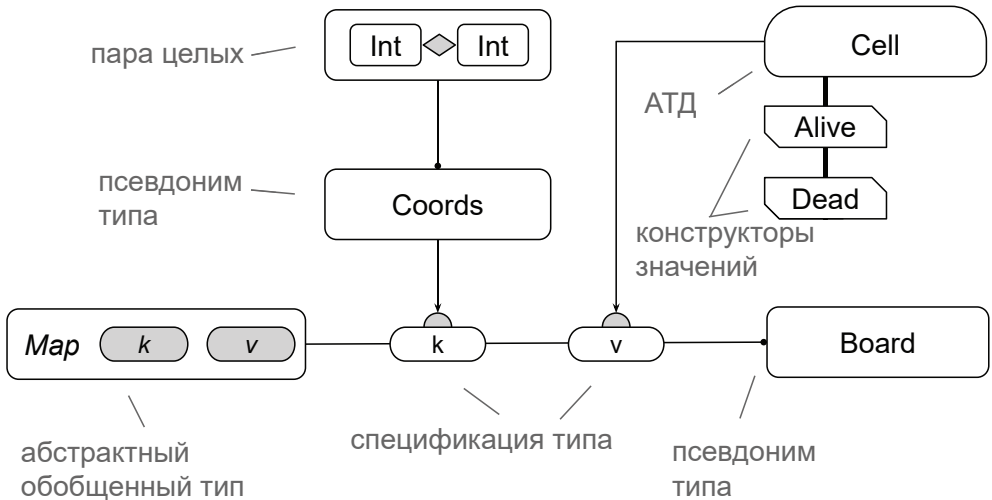


Рис. 1.2. Диаграмма типизированных форм для Board

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.

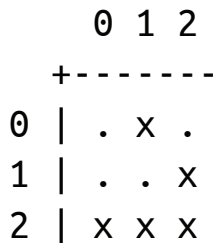


Рис. 1.3. Планер

Первоначальный код приложения может выглядеть как функция `main`, в которой сосредоточены все операции чтения-записи файлов и кое-какие зашифрованные 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».

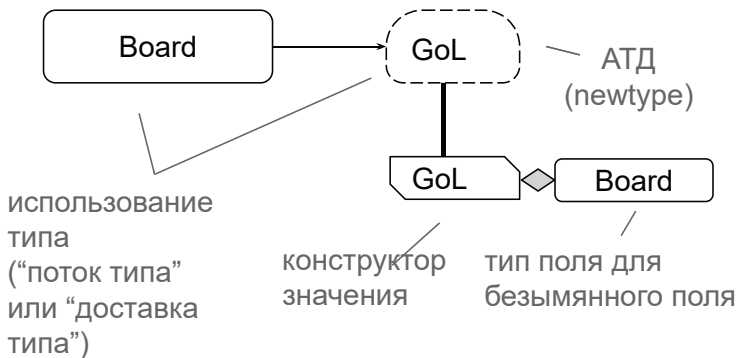


Рис. 1.4. Диаграмма типизированных форм для игры «Жизнь»

Добавление еще одной `newtype`-обертки для другого автомата, `Seeds`,

```
newtype Seeds = Seeds Board
```

позволяет различить оба типа (подобных `GoL` и `Seeds`) на этапе компиляции. Теперь их нельзя взаимозаменять, даже если хранимые игровые доски идентичны.

Стандартный пример – имя и фамилия человека. Типом должна быть строка, но типы строк используются в проектах повсеместно и означают много разных предметов. Поэтому мы инкапсулируем типы с помощью `newtype`:

```
newtype FirstName = FirstName String
newtype LastName = LastName String
```

Теперь трудно спутать эти типы с типом пути к файлу `FilePath`, который тоже является синонимом `String`:

```
type FilePath = String

lastName :: LastName
lastName = LastName "O'Neill"

lastFileName :: FilePath
lastFileName = "last_file_name.txt"

main = writeFile lastName "McCarthy" -- ошибка компиляции
```

Этот прием поможет нам спасти некоторые миры от уничтожения ценой небольшого неудобства и шаблонного кода (`boilerplate code`). Нам придется распаковать `newtype`, чтобы добраться до его содержимого:

```
writelastName :: FilePath -> LastName -> IO ()
writelastName path (LastName name) #A
  = writeFile path name

main = writelastName lastFileName lastName

#A Распаковка путем сопоставления с образцом
```

Передать `FirstName` в функцию `writelastName` невозможно, и без дополнительных механизмов код получается слишком специализированным и негибким.

Одно лишь это средство (`newtype`) не позволяет далеко уйти от обычного кода с базовыми типами и значениями. Все наши рассуждения по-прежнему вращаются вокруг преобразования значений во время выполнения и никаким образом не касаются модификации значений на этапе компиляции. Мы не преобразуем типы в значения и обратно, не преобразуем одни типы в другие – а это именно те операции, которые составляют ядро проектирования на уровне типов. В нашем случае типы описывают, а значения трудятся – и ничего более.

С этим старым добрым стилем можно сделать все. Нет нужды вносить дополнительную семантику в типы; просто вызывайте функции, обрабатывайте данные, выражайте сущности в виде абстрактных типов данных (АТД) и наслаждайтесь всеми благами обычного функционального программирования. Если вам нужны рекомендации по применению такого подхода, следуйте моей методологии *Functional Declarative Design (FDD)* из книги «*Functional Design and Architecture*», где предложено много полезных идей для улучшения функционального кода.

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

Также есть ознакомительная статья о методологии:

ССЫЛКА. Alexander Granin «Functional Declarative Design: A Comprehensive Methodology for Statically-Typed Functional Programming Languages» <https://github.com/graninas/functional-declarative-design-methodology>

Преобразования типов в другие типы, а потом и в значения делают код еще более абстрактным, часто более сложным, так что он теряет очевидность. Во всех языках средства уровня типов приносят существенные синтаксические и семантические издержки. Решение перейти на уровень типов очень ответственное и должно быть хорошо обосновано. Должна быть убедительная причина и какая-то разумная цель, отражающая реальные, а не воображаемые потребности бизнеса. Иными словами, программирование на типах должно быть прагматичным, иначе мы рискуем внести лишнюю сложность без значимых преимуществ.

1.1.2. Прагматичное проектирование на уровне типов

Ресурсов, посвященных типам, немало. Многие пытаются ответить на вопрос: «Почему программирование на типах – это круто?» Иногда вас пытаются учить математике, а не созданию полезных программ. Авторы с удовольствием демонстрируют красоту трюков с типами и призывают вас разделить их энтузиазм по поводу интеллектуальных игр вокруг типов.

Прагматизм – это о другом.

Он ищет ответы на такие вопросы: «Как применить проектирование на уровне типов к реальным задачам?», «Как достичь бизнес-целей и сохранить простоту кода?», «Почему вот это решение в типах лучше решения в обычных значениях?». Вот мои варианты того, когда проектирование на уровне типов помогает:

- можно лучше выразить особо ответственные предметные области;
- меньше ошибок и больше гарантий правильности кода;
- меньше необходимости в обширном тестировании;
- лучше расширяемый код;
- обобщенный код с большим потенциалом повторного использования;
- вычисления на этапе компиляции;
- генерирование кода;
- оптимизация производительности и бесплатные абстракции.

Подход разумен тогда, когда можно получить некий значимый, ощутимо лучший результат при использовании кода на уровне типов, чем без него. Внесенная сложность при этом все еще может нанести ответный удар, и хорошо бы, чтобы потенциальный рефакторинг не превращался в переписывание всего приложения. Такие решения зачастую трудно реализовать и понять, и спустя некоторое время в них уже не сможет разобраться даже сам автор. А иногда вы даже ощущаете себя героем рассказа «Цветы для Эдждернона» – когда его интеллект угас, и он перестал понимать написанное им же самим.

Моя методология прагматичного проектирования на уровне типов (*Pragmatic Type-Level Design, PTLD*) решает эти проблемы. Она призывает управлять искусственной сложностью и взвешивать все за и против тех или иных инструментов и подходов.

ОПРЕДЕЛЕНИЕ. *Естественная сложность* (essential complexity): сложность, внутренне присущая и неотделимая от предметной области, отражающая его сущность. Невозможно программировать предметную область, вообще не принося в проект естественную сложность.

ОПРЕДЕЛЕНИЕ. *Искусственная сложность* (accidental complexity): сложность, обусловленная инструментами и проектными решениями, используемыми при реализации предметной области. Искусственная сложность решений и подходов в данном контексте может изменяться. Для одних решений искусственная сложность выше, для других – при схожих обстоятельствах – ниже. Чем больше искусственной сложности в проекте, тем дороже его развивать и тем выше риск провала по техническим причинам.

Искусственная сложность – главный враг проектирования ПО, и в том числе проектирования на уровне типов. Сверхцель архитекторов ПО – управление сложностью и снижение ее до приемлемого уровня, при котором проект не закончится крахом в долгосрочной перспективе. Искусственная сложность имеет свойство расти, если ничего не предпринимать для ее ограничения. Рискну утверждать, что чем больше средств уровня типов используется, тем быстрее растет искусственная сложность. На рис. 1.5 это показано на графике.

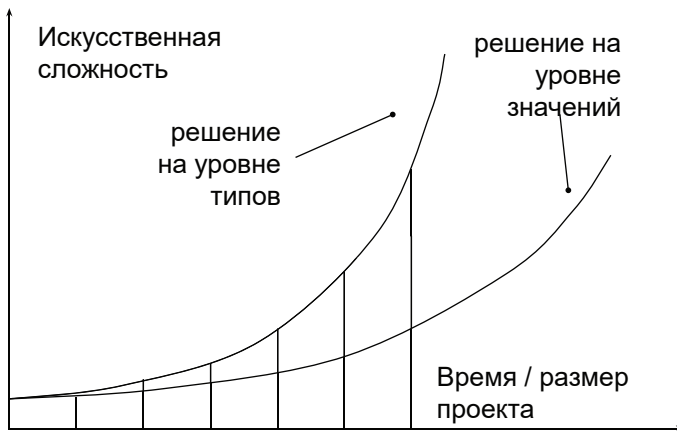


Рис. 1.5. Искусственная сложность решений на уровне типов и на уровне значений

Продвигаться вперед следует осторожно, потому что программирование на типах сопряжено с различными рисками.

- *Переусложнение системы.* Выбранные проектные решения характеризуются несоразмерно высокой искусственной сложностью, которая приведет к выходу за рамки бюджета и негативно отразится на состоянии проекта в долгосрочной перспективе.

- *Переусложнение кода.* Переусложненный код трудно понять и модифицировать из-за неудачного выбора имен и включения слишком абстрактных и общих решений. Часто это приводит к размыванию предметной области в трясине вспомогательных механизмов и, как следствие, к полной невозможности выявить понятия и поведение предметной области на основе изучения кода.
- *Неверные проектные решения.* В программировании на типах существует много способов добиться одной и той же цели. Неудачные решения внезапно могут стать препятствием на пути развития кода. Из-за этого, возможно, придется полностью переписывать значительные части проекта или прибегать к «грязным трюкам», что только ухудшит ситуацию.
- *Желание потешить свое эго.* Программирование на типах, обобщенное программирование и метапрограммирование могут быть весьма притягательны для людей, сосредоточенных на интеллектуальном наслаждении и забывших о том, для чего их принимали на работу. Очень легко подменить реальные цели желанием продемонстрировать собственную изобретательность и способность жонглировать высокоуровневыми сложными идеями. И все же нам, программистам, надлежит всегда блюсти честность и техническую корректность при обосновании решений.

Чтобы избежать этих рисков и сохранить рациональность, я предлагаю следующие общие принципы проектирования в рамках методологии PTLD.

ПРИНЦИП I – *бритва Оккама.* Лучше ограничиться набором средств, покрывающих 95 % случаев, и не вводить дополнительные средства без острой необходимости.

ПРИНЦИП II – *не надо перфекционизма.* Нет необходимости делать код стопроцентно совершенным, корректным и красивым. Если повсеместное применение типов раздувает сложность кода, то лучше обратиться к другим, менее красивым решениям.

ПРИНЦИП III – *невзрачно, но единообразно.* Единый, пусть даже невзрачный подход лучше, чем новое и непохожее решение каждой следующей задачи.

ПРИНЦИП IV – *прагматизм.* Красота и математическая корректность не столь ценны, как несовершенная, но работающая система. Можно платить срезанием углов и компромиссами за снижение искусственной сложности. Проектирование всегда должно преследовать реальные бизнес-цели.

Таким образом, прагматичное проектирование на уровне типов – луч счета в темном царстве, карта, которая приведет вас к сокровищам в диком мире, полном пугающих тварей и опасных соблазнов.

На рис. 1.6 приведено сравнение трех методологий: объектно ориентированного проектирования (Object-Oriented Design, OOD), функционального декларативного проектирования (FDD) и прагматичного проектирования на уровне типов (PTLD).

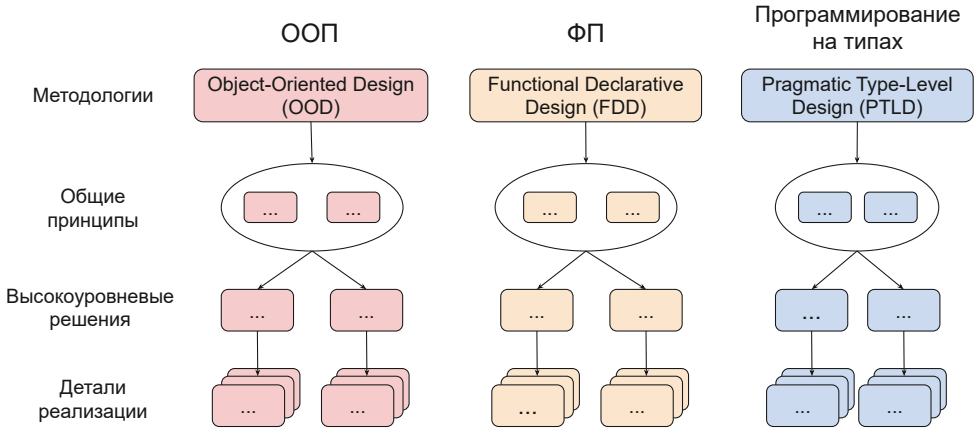


Рис. 1.6. Три методологии проектирования программного обеспечения

Важно, что все принципы проектирования, известные нам по OOD и FDD, вполне могут пригодиться и для PTLD. Мы встретимся со многими из них: принципы SOLID, низкая связанность и высокая сцепленность, разделяй и властвуй и т. д. Диаграмма (рис. 1.6) также говорит нам, что зачастую из этих принципов непосредственно вытекают высокоуровневые технические решения, и только потом определяются детали реализации. На рис. 1.7 показаны принципы и методологии.

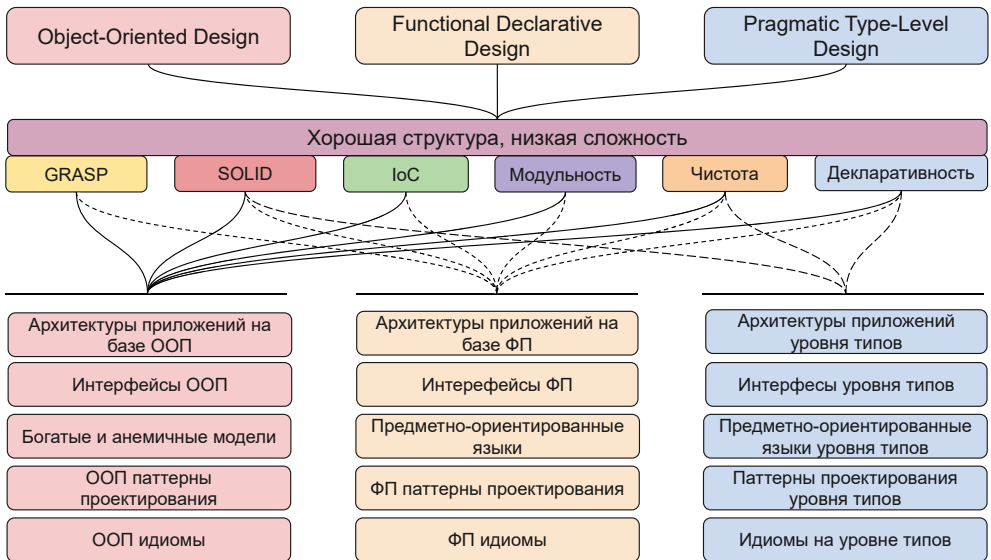


Рис. 1.7. Принципы проектирования и идеи, позволяющие контролировать сложность

Я с особенным нетерпением предвкушаю обсуждение предметно-ориентированных языков на уровне типов, потому что это мой любимый способ выразить специфику предметной области и вместе с тем одолеть искусственную

сложность. Взгляните на следующий код пока без объяснений; это полноценный тип, а не просто какое-то обычные значение:

```
type GameOfLifeStep = Step
  [ StateTransition 0 1 [ CellsCount 1 [3]] -- "Born"
  , StateTransition 1 1 [ CellsCount 1 [2,3]] -- "Survive"
  , DefaultTransition 0
  ]
```

Тип, выраженный средствами Haskell, описывает логику клеточного автомата исключительно с помощью других типов. Со временем мы доберемся и до этого. Но прежде нужно изучить некоторые идеи и приемы работы на уровне типов.

1.1.3. Диаграммы типизированных форм

Научиться работать на уровне типов нелегко, я это говорю из собственного опыта. Немудрено заблудиться в определениях и взаимосвязях, особенно когда синтаксис языка программирования сложен. Haskell здесь, конечно, не C++, но иногда запутанность синтаксиса и семантики превосходит все разумные пределы.

Я разработал специальный визуальный язык структурных диаграмм для своей методологии PTLD. Я называю его типизированными формами и уверен, что он хорошо послужит двум целям: прояснить смысл объяснений и сделать книгу эстетически привлекательной. Я научу вас этому языку, когда придет время. Можете также обратиться к полному описанию в приложении А «Диаграммы типизированных форм».

Ниже приведен пример такой диаграммы.

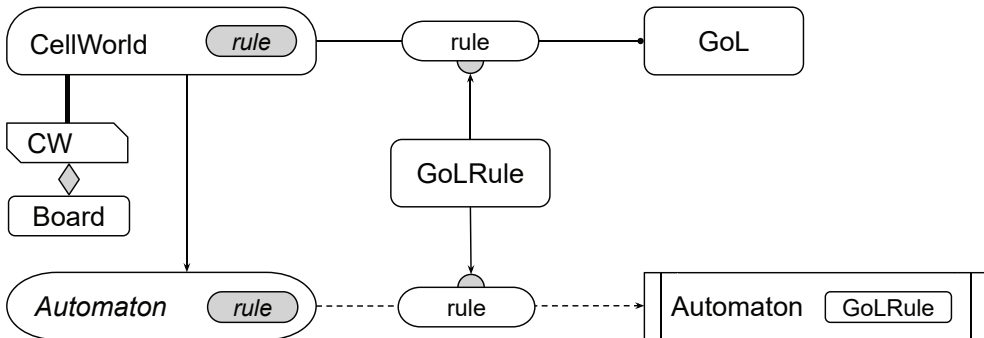


Рис. 1.8. Диаграмма типизированных форм

Симпатично, правда?

1.2. В ПРЕДВКУШЕНИИ ПРОЕКТИРОВАНИЯ НА УРОВНЕ ТИПОВ

В старом добром Haskell 2010 введено много средств уровня типов: определения типов (type definitions), классы типов (type classes), типовые переменные (type variables), ограничения (constratins), роды (kinds), фантомные типы (phantom types) и алгебраические типы данных (algebraic data types). Благода-

ря десяткам расширений компилятора GHC язык стал еще мощнее: литералы уровня типов (type-level literals), семейства типов (type families), многопараметрические классы типов (multiparam type classes), обобщенные алгебраические типы данных (generalized algebraic data types), различные варианты полиморфизма на уровне типов и т. д. и т. п. – сложно даже перечислить все эти средства, что уж говорить о комбинаторном росте числа их сочетаний. Именно поэтому я считаю программирование на типах чистым искусством, которым каждый владеет по-своему. Мы превратим это искусство в инженерную дисциплину, единообразно применимую к различным задачам. Но сначала кое-какие основы.

Я продемонстрирую два важных средства, лежащих в основе проектирования на уровне типов: классы типов и типы-литералы. Многие другие средства будут введены по мере необходимости.

1.2.1. Основы обобщенных типов

Вернемся к игре «Жизнь». У меня для вас новости: функциональные требования внезапно изменились, как часто бывает в реальности. Теперь программа должна поддерживать все клеточные автоматы, а не только тот, который придумал и описал Джон Конвей.

Вы помните тип `GoL`, который есть обертка типа `Board`? Теперь у нас есть три таких типа для разных правил:

```
newtype GoL = GoL Board #A
newtype Seeds = Seeds Board
newtype Replicator = Replicator Board
```

#A Различные правила «жизнеподобных» игр: `Game of Life`, `Seeds`, `Replicator`

Все три описывают правила, похожие на те, что действуют для «Жизни», т. е. работают на двумерной сетке клеток с двумя состояниями и определены на одной и той же 8-клеточной области непосредственных соседей. Правила включения и выключения клеток различаются, и вообще-то мы не хотим, чтобы правило `Seeds` случайно было применено к миру `Replicator`. С помощью ключевого слова `newtype` мы можем предотвратить такую неразбериху, но должны завести отдельные функции шага для каждого типа:

```
golStep :: GoL -> GoL
seedsStep :: Seeds -> Seeds
replicatorStep :: Replicator -> Replicator
```

Кое-какую безопасность мы обеспечили, включив дополнительное знание в исходный тип `Board`. Улучшение было простым и изящным, а компилятору без разницы: одним `newtype` больше, одним меньше. Но мы-то не машины. Существует $2^{18} = 262144$ потенциальных «жизнеподобных» правил, и лишь малая толика их исследована. Представьте, что нам придется добавлять все новые и новые обертки, стараясь угнаться за неистощимым воображением поклонников «Жизни»:

```
newtype Diamoeba = Diamoeba Board #A
newtype HighLife = HighLife Board
newtype DayAndNight = DayAndNight Board
```

```
diamoebaStep :: Diamoeba -> Diamoeba
highLifeStep :: HighLife -> HighLife
dayAndNightStep :: DayAndNight -> DayAndNight
```

#A Еще три «жизнеподобных» правила: **Diamoeba**, **HighLife**, **DayAndNight**

Ох, безопасность безопасностью, но какой же это шаблонный код (boilerplate code)! Функции итерирования тоже множатся:

```
iterateGoL :: Int -> GoL -> GoL
iterateReplicator :: Int -> Replicator -> Replicator
...
```

При ближайшем рассмотрении все эти функции одинаковы, если предположить, что каждая из них пользуется функцией, реализующей один шаг конкретного автомата:

```
golStep :: GoL -> GoL #A
golStep = ... #B

iterateGoL :: Int -> GoL -> GoL
iterateGoL n gol | n == 0 = gol
iterateGoL n gol | n > 0 =
  head #C
  (drop n #D
   (iterate golStep gol)) #E
iterateGoL _ _ = error "Invalid iteration count"

#A Шаг «Жизни»
#B Правило «Жизни»
#C Взять шестой элемент списка итерированных миров
#D Удалить первые n элементов из списка итерированных миров
#E Бесконечный список итерированных миров; вызов функции шага
```

Функция `iterateReplicator` имеет точно такое же тело с точностью до идентификатора. Тут-то и приходят на помощь обобщенные типы. Мы можем перенести всю эту логику в абстрактную функцию, которая принимает функцию шага любого клеточного автомата и итерирует заданный мир по заданному правилу `n` раз:

```
iterateWorld :: (ca -> ca) -> Int -> ca -> ca
iterateWorld step n world | n == 0 = world
iterateWorld step n world | n > 0 =
  head (drop n (iterate step world))
iterateWorld _ _ _ = error "Invalid iteration count"
```

Неважно, что представляет собой тип `ca`, коль скоро в нем имеется функция `step`. Нам нет нужды знать о внутренней структуре фактического типа `ca`, потому что это знание инкапсулировано и используется внутри функции `step`. Мы даже обеспечили кое-какую типобезопасность, поскольку невозможно применить функцию шага не к тому миру:

```
glider' :: GoL
glider' = GoL glider

> iterateWorld golStep 5 glider' -- OK
> iterateWorld seedsStep 5 glider' -- ошибка компиляции
```

ОПРЕДЕЛЕНИЕ. *Типобезопасность*: невозможность выполнить неверное преобразование типов, сконструировать недопустимый тип или скомбинировать типы и значения, которые комбинироваться не должны.

Это работает, потому что все четыре типовых переменных `ca` (`ca` означает «cellular automaton», клеточный автомат) в одном вызове `iterateWorld` должны быть одного типа.

ПОДСКАЗКА. В других языках тоже есть обобщенные типы. Дополнительную информацию см. в части IV «Розеттский камень».

Что делать с другими полезными функциями, которые должны знать о внутреннем устройстве? Вот функция сохранения:

```
saveToFile :: FilePath -> Board -> IO ()
```

Конечно, мы не хотим плодить много методов вроде `saveGoLToFile`, `saveSeedToFile` и т. д., но и обобщить пока тоже не можем:

```
saveToFile' :: FilePath -> ca -> IO ()
```

Мы знаем, что все «новые типы» (`newtypes`) обертывают один и тот же тип `Board`, но эта функция ничего про их структуру не знает и видеть сквозь типовую переменную `ca` не может. Значение этого произвольного типа `ca` непрозрачно для функции, и нет никаких указаний на то, чем `ca` будет после полной спецификации. Это значит, что наше обобщение ограничено, и, стало быть, нужно искать другой путь.

1.2.2. Основы классов типов

Классы типов позволяют обобщать функции автомата более строго. Пока что мы отметили следующие элементы любого автомата:

- тип, отличающий один автомат от другого;
- конкретная функция шага, которая выполняет соответствующее правило.

Мы выяснили, что, для того чтобы функция `saveToFile` могла работать, необходимо развернуть тип автомата, выделив из него значение `Board`. Это можно сделать довольно просто:

```
saveToFile'' :: (ca -> Board) -> FilePath -> ca -> IO ()
saveToFile'' unwrap file world = saveToFile file (unwrap world)
```

И затем:

```
unwrapSeeds :: Seeds -> Board
unwrapSeeds (Seeds world) = world
```

```
seedsWorld :: Seeds
seedsWorld = Seeds glider
```

```
> saveToFile'' unwrapSeeds "seeds.txt" seedsWorld
```

Однако хотя этот дизайн вполне допустим, я склонен считать, что он оставляет нас на уровне значений и вынуждает выполнять все эти вспомогательные

функции. Однако теперь мы готовы ввести интерфейс класса типов для всех «жизнеподобных» клеточных автоматов. В нем будет всего три метода.

Листинг 1.1. Интерфейс класса типов для клеточных автоматов

```
class Automaton a where
  step :: a -> a
  wrap :: Board -> a
  unwrap :: a -> Board
```

Мы реализуем его по-разному для «новых типов» `GoL`, `Seeds`, `Replicator` и т. д. И это совершенно правильно: типы разные, поэтому для каждого должен быть свой экземпляр этого класса типов. Если бы мы попытались несколько раз инстанцировать класс типов с `Board`, то потерпели бы неудачу.

ПОДСКАЗКА. Экземпляры классов типов конфликтуют, если они изготовлены для одного и того же типа и импортированы в один и тот же модуль. У инстанцирования класса типов есть и другие подводные камни, о которых вы можете прочитать в других источниках.

ССЫЛКА. Vitaly Bragilevsky «Haskell in Depth», <https://www.manning.com/books/haskell-in-depth>

ПОДСКАЗКА. В Rust, C++, Scala, PureScript и некоторых других языках есть механизмы, похожие на классы типов. Дополнительную информацию см. в части IV «Розеттский камень».

Листинг 1.2. Две разные реализации класса типов

```
-- module GameOfLife:

newtype GoL = GoL Board

instance Automaton GoL where #A
  step = goLStep
  wrap board = GoL board
  unwrap (GoL board) = board

-- module Seeds:

newtype Seeds = Seeds Board

instance Automaton Seeds where #B
  step = seedsStep
  wrap board = Seeds board
  unwrap (Seeds board) = board

#A Реализация для GameOfLife
#B Реализация для Seeds
```

Диаграмма типизированных форм представлена на рис. 1.9.

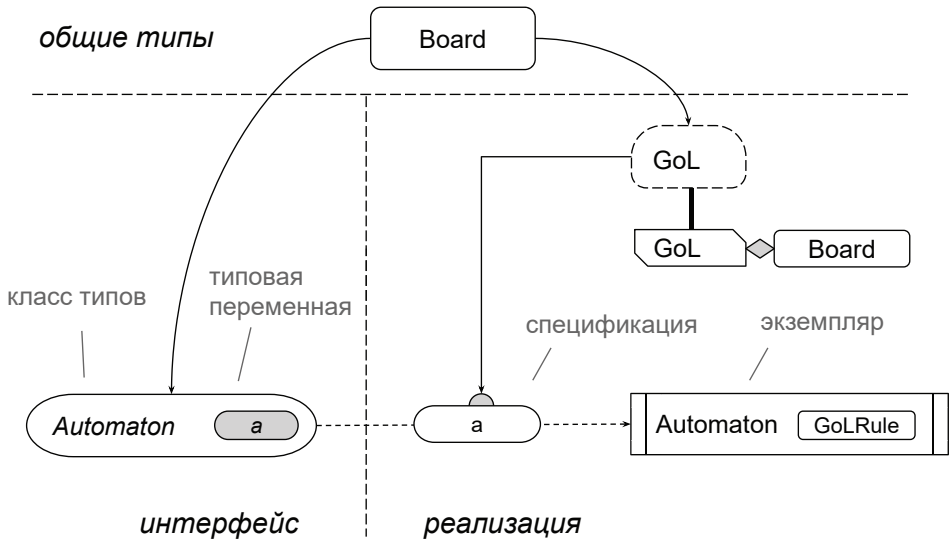


Рис. 1.9. Класс типов и его экземпляр

С нашими обобщенными функциями стало удобнее работать, потому что больше не нужно передавать вспомогательные функции. Все необходимое можно запросить у конкретного экземпляра типа классов `Automaton`, и наше право на это оформлено в объявлении типа функции:

```
iterateWorld :: Automaton ca => Int -> ca -> ca
loadFromFile :: Automaton ca => FilePath -> IO ca
saveToFile :: Automaton ca => FilePath -> ca -> IO ()
```

Все эти функции знают, что фактический тип, подставленный вместо `ca`, обязательно реализует функции `step`, `wrap` и `unwrap`, поэтому допустимо вызывать их при любой необходимости:

```
loadFromFile :: Automaton ca => FilePath -> IO ca
loadFromFile file = do
  (board :: Board) <- loadBoardFromFile file
  pure (wrap board) #A
```

#A Метод 'wrap' из класса типов

Наконец, мы можем явно указать, с какой реализацией имеем дело. В следующем фрагменте компилятор выведет, что функции должны браться из экземпляра, связанного с типом `GoL`:

```
gol1 :: GoL #A
  <- loadFromFile "./data/GoL/glider.txt" #B

let gol2 = iterateWorld 5 gol1

saveToFile "./data/GoL/glider_5th_gen.txt" gol2
```

#A Явная спецификация типа

#B Вызов обобщенной функции

Именно так классы типов обеспечивают расширяемость. Теперь мы можем добавлять новые клеточные автоматы, не изменяя обобщенные функции. Новый экземпляр даже не обязан быть `newtype`, обертывающим `Board`. Например, я мог бы представить свой мир ассоциативным списком вместо словаря:

```
data Diamoeba = Diamoeba
  { diamoebaBoard :: [(Int, Int), Cell]
  , diamoebaWorldName :: String
  }
```

Заодно я добавил в этот ADT еще одно поле, содержащее имя мира. Оно не создаст препятствий механизму `Automaton`, хотя и толку от него немного. Мы должны быть готовы к тому, что в конечном итоге имя потеряется или будет проигнорировано:

```
instance Automaton Diamoeba where
  step = diamoebaStep
  wrap board = Diamoeba (Map.toList board) "" #A
  unwrap (Diamoeba board _) = Map.fromList board #B
```

#A По умолчанию имя пусто

#B Имя потеряно

В остальном дизайн кажется вполне приличным. Но так ли это? Не совсем. В нем присутствуют более серьезные изъяны и ограничения; более того, он даже нарушает два принципа проектирования, что не есть хорошо. Давайте поговорим об этом.

1.2.3. Принципы проектирования

Еще раз взгляните на этот крохотный, так невинно выглядящий класс типов. Как этот милый малыш может что-то нарушить?

```
class Automaton a where
  step :: a -> a
  wrap :: Board -> a
  unwrap :: a -> Board
```

А вот поди ж ты – он нарушает сразу два принципа SOLID: *принцип единственной ответственности* (*single responsibility principle – SRP*) и *принцип разделения интерфейсов* (*interface segregation principle – ISP*).

SRP говорит, что программная единица должна иметь только одну обязанность. А у `Automaton` их две: эволюция мира с помощью `step` и доступ к значению `Board` с помощью `wrap` и `unwrap`. А что насчет этих двух? Нахождение их в классе типов нарушает принцип ISP, потому что они не относятся к предметной области. Этот принцип наставляет нас группировать обязанности в отдельных интерфейсах, а не смешивать разные уровни абстракции, как здесь.

Принципы проектирования, в частности SOLID, позволяют обнаруживать «запахи» в коде и разумно рассуждать о проблемах. Вопреки распространенному заблуждению (утвердившемуся в сообществе разработчиков) принципы SOLID, впервые сформулированные Робертом Мартином, не голая теория; они практичны и полезны.

В процессе проектирования мы не просто ваяем код – это было бы неумной и антиинженерной практикой. Напротив, мы руководствуемся общими верхнеуровневыми аргументами, чтобы оценить, насколько хороша наша идея. Принципы SOLID и некоторые другие, например «низкая связанность, высокая сцепленность», будучи применены правильно, улучшают качество кода в целом.

Принцип единственной обязанности (single responsibility principle, SRP). Сущность должна в каждый момент времени отвечать за что-то одно.

Принцип открытости-закрытости (open-close principle, OCP). Система должна иметь стабильные интерфейсы, и добавление новых реализаций или нового поведения не должно оказывать влияния на уже согласованные контракты о поведении (открытость для расширения, закрытость для модификации).

Принцип подстановки Лисков (Liskov substitution principle, LSP). Реализации можно динамически подменять, и это не приводит к поломке клиентского кода.

Принцип разделения интерфейсов (interface segregation principle, ISP). Обязанности должны быть распределены между интерфейсами, в достаточной мере сфокусированными и консистентными.

Принцип инверсии зависимостей (dependency inversion principle, DIP) утверждает, что бизнес-логика должна зависеть только от интерфейсов, а не от фактической реализации, предоставляемой позже в интерпретирующем процессе.

Низкая связанность, высокая сцепленность (low coupling, high cohesion). Сущность должна зависеть от как можно меньшего числа внешних, не связанных между собой сущностей (низкая связанность). Она должна содержать только то, что необходимо, и не должна иметь дополнительных, не свойственных ей обязанностей (высокая сцепленность). Этот принцип является частью еще одного набора принципов – GRASP (General Responsibility Assignment Software Patterns – общие программные паттерны распределения обязанностей).

Эти принципы первоначально были сформулированы для объектно ориентированного проектирования, но на самом деле они универсальны. В книге FDaA я продемонстрировал, что они применимы к функциональному программированию и придают моей методологии FDD функциональную насыщенность и действенность. Разумеется, без этих принципов проектирование на уровне типов не было бы прагматичным, и впоследствии вы увидите, как они применяются к типам.

Вернемся к подходу на основе newtype, потому что он не оптимален и несет с собой слишком много шаблонного кода. Мы исправим это, введя новый тип мира и воспользовавшись интересным средством: литералами уровня типа. Новый дизайн позволит улучшить механизм Automaton в следующей главе.

1.2.4. Основы литералов уровня типа

Показанный ранее дизайн обладает хорошим свойством: мы можем иметь мир с любой внутренней структурой. Например, это мог бы быть подобный словарю тип Board:

```
newtype GameOfLife = GameOfLife Board
newtype Replicator = Replicator Board
```

Или двумерный вещественный массив:

```
import Data.Vector
data Diamoeba = Diamoeba
  { diamoebaBoard :: Vector (Vector Cell) #A
  , diamoebaWorldName :: String
  }
```

#A Двумерный массив

Ничего плохого в этой возможности нет, но ее, скорее, можно назвать обычным дизайном, а никак не дизайном на уровне типов. Продвигаясь дальше, мы жертвуем возможностью иметь разную структуру мира для разных автоматов. И будем избегать заведения `newtype` для каждого нужного нам мира. Будет только один тип для всех автоматов:

```
newtype CellWorld = CW Board
```

Если мы попытаемся изготовить фактический автомат с помощью одних лишь синонимов типов, то столкнемся с проблемой.

```
type GoL = CellWorld
type Seeds = CellWorld
```

Эти два типа в точности одинаковы, мы имеем всего лишь два разных идентификатора одного и того же типа. И с классом типов `Automaton` это работать не будет, потому что нельзя создать один экземпляр дважды. Они должны быть существенно различны. Один из способов добиться желаемого – параметризация на уровне типов, например с помощью типов-строк. Итак, у нас будет параметризуемый шаблон `CellWorld` и конкретные типы автоматов, различающихся на уровне типов.

Во-первых, вот как теперь выглядят конкретные миры:

```
{-# LANGUAGE DataKinds #-}

type GoLRule = "Game of Life" #A
type SeedsRule = "Seeds"

type GoL = CellWorld GoLRule
type Seeds = CellWorld SeedsRule
```

#A Здесь используются `DataKinds`

ПОДСКАЗКА. Типы-строки – это типы, и они различаются, если различно содержимое.

ПОДСКАЗКА. Типы-строки требуют, чтобы было активировано расширение языка `DataKinds`. В этом случае компилятор рассматривает обычные строки как типы в соответствующих местах.

ПОДСКАЗКА. Типы-строки и другие литералы уровня типов не уникальная черта Haskell. Дополнительную информацию см. в части IV «Розеттский камень».

Во-вторых, мы должны перепроектировать тип `CellWorld`. Придание ему только одного параметра произвольного типа работает лишь частично:

```
newtype CellWorld arbitraryType = CW Board

type SomeIntWorld = CellWorld Int
```

Здесь `arbitraryType` может содержать только регулярные типы: `Int`, `Char` и т. д. Он не примет типы `"Game of Life"` и `"Seeds"` из представленного выше кода. Эти строки не регулярные типы, а специализированные типы-строки. В Haskell это различие важно и связано с понятием рода (тип типа, `kind`). Забегая вперед, скажем, что `Int`, `Char` и другие обычные типы имеют род `*` (звездочка) (подробнее об этом в следующих главах), тогда как типы-строки имеют род `Symbol`. На псевдокоде:

```
type Int :: kind * #A
type Char :: kind *
type "Game of Life" :: kind Symbol #B
type "Seeds" :: kind Symbol
```

#A Обыкновенный род для регулярных типов
#B Специфичный (уникальный) род

Полное описание типа `CellWorld`, приведенного выше, включает ранее опущенную факультативную спецификацию рода:

```
newtype CellWorld (arbitraryType :: *) = CW Board
type IntWorld = CellWorld Int -- ОК: Int имеет род *
type SeedsRule = CellWorld "Seeds" -- Ошибка: "Seeds" – Symbol, а не *
```

Мы должны указать род `Symbol` в определении `CellWorld` по двум причинам: это вид документации, и компилятор должен это знать, чтобы правильно произвести вывод.

```
{-# LANGUAGE DataKinds #-}
{-# LANGUAGE KindSignatures #-}

import GHC.TypeLits ( Symbol )

newtype CellWorld (rule :: Symbol) #A
  = CW Board
```

#A Здесь используются расширения языка `DataKinds` и `KindSignatures`

Ниже показана соответствующая диаграмма.

ПРИМЕЧАНИЕ. В современных расширениях Haskell родов больше нет, они унифицированы идеей «типа типов». Специальный род `Type` пришел на смену символу звездочки, а различие между типами и родами устранено. В результате система типов в целом стала еще более запутанной. Чтобы избежать закипания мозга, мы пока будем придерживаться старой терминологии.

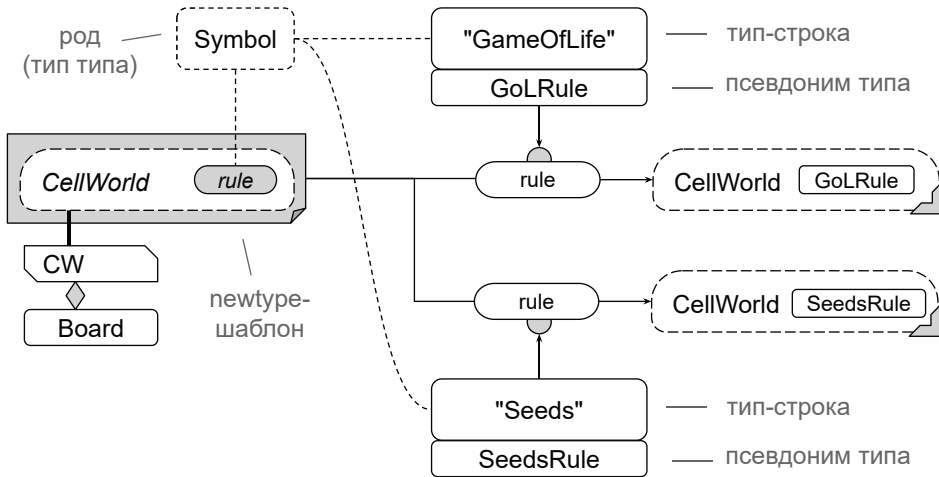


Рис. 1.10. Диаграмма типизированных форм для CellWorld

ПРИМЕЧАНИЕ. Два расширения, `DataKinds` и `KindSignatures`, отвечают за форму спецификации (`rule :: XYZ`). Проверим:

```
{-# LANGUAGE KindSignatures #-}
{-# LANGUAGE DataKinds #-}

import GHC.TypeLits (Symbol)

data StandardHaskell a = Test0
data NeedKindSignatures (a :: *) = Test1
data NeedKindSignaturesAndDataKinds (a :: Symbol) = Test2
```

У нас нет иного выбора, кроме как последовать этому рецепту без подробных объяснений. Мы вернемся к этим расширениям и объявлениям позже.

ПРИМЕЧАНИЕ. У типа `CellWorld` есть параметр `rule`, но в конструкторе значений `CW` он не используется. Поэтому `rule` оказывается фантомным типом; аналогично `iAmPhantom` является фантомным типом в объявлении ниже:

```
data SomeType iAmPhantom = NothingToSeeHere
```

Фантомные типы несут дополнительную информацию о других типах, в которых свободные неиспользуемые типы-параметры находят применение. В последующих главах мы увидим, что фантомные типы важны в проектировании на уровне типов.

Что еще можно делать с типизированными строками? Мы включили имя автомата, и было бы хорошо иметь возможность напечатать его, коль скоро задан какой-то мир. Такой переход от типов к значениям в Haskell прост, но нуждается в помощи со стороны нескольких экзотических существ из глубин языка. Зоопарк состоит из:

- класса типов `KnownSymbol` из модуля `GHC.TypeLits`;
- функции `stringVal`, принадлежащей классу типов `KnownSymbol`.

Следующая функция производит преобразование, получив значение типа `CellWorld`:

```
import GHC.TypeLits (Symbol, KnownSymbol, symbolVal)

automatonWorldName
  :: KnownSymbol rule #A
  => CellWorld rule #B
  -> String
automatonWorldName world = symbolVal world
```

#A Ограничение требует, чтобы 'rule' имел род `Symbol`
#B Обозначать 'rule' как `Symbol` необязательно

Указывать род `Symbol` необязательно, потому что он уже присутствует в определении `KnownSymbol` и `CellWorld`. Но это допускается:

```
automatonWorldName
  :: KnownSymbol (rule :: Symbol)
  => CellWorld (rule :: Symbol)
  -> String
automatonWorldName world = symbolVal world
```

Функции `symbolVal` неинтересно само значение, но она нуждается в фактическом типе строки, который сопутствует переменной `world`. Переменная `world` служит своего рода прокси необходимого типа для `symbolVal`, чтобы та знала, какой тип-строку мы хотим преобразовать:

```
symbolVal :: KnownSymbol n => proxy n -> String
```

Функция `automatonWorldName` смотрит на тип `CellWorld` и видит, что `rule` имеет род `Symbol`, который пригоден для класса типов `KnownSymbol`, а значит, и для `symbolVal`.

Использовать ее просто при условии, что имеется мир; например, игра «Жизнь»:

```
type GoLRule = "Game of Life"
type GoL = CellWorld GoLRule

gameOfLifeEmptyWorld :: GoL
gameOfLifeEmptyWorld = CW Map.empty

main :: IO ()
main = print (automatonWorldName gameOfLifeEmptyWorld)
> "Game of Life"
```

Haskell поддерживает также натуральные типы-числа (от 0 до бесконечности). Мы легко могли бы заменить ими типы-строки, добившись похожего эффекта.

```
import GHC.TypeLits (Nat)

newtype IndexedCellWorld (index :: Nat) = ICW Board
```

Существует род `Nat`, класс типов `KnownNat` и функция, которая преобразует типы-числа в целые уровня значений:

```
import GHC.TypeLits (Nat, KnownNat, natVal)
import Data.Proxy (Proxy (..))

type FortyTwo = (42 :: Nat)

> natVal (Proxy :: Proxy FortyTwo)
> 42
```

Здесь мы имеем только тип `FortyTwo` и ничего более. В отсутствие реального значения этого типа мы конструируем значение типа `Proxy` из модуля `Data.Proxy`, которое поможет `natVal` понять, что мы имеем в виду. `symbolVal` также поддерживает этот трюк:

```
type GameOfLifeRule = ("Game of Life" :: Symbol)
> symbolVal (Proxy :: Proxy GameOfLifeRule)
> "Game of Life"
```

Помимо строк и чисел, существуют типы-символы, типы – булевы значения, алгебраические типы данных уровня типов и т. д. Все это нам понадобится, но давайте действовать постепенно, шаг за шагом.

Итак, приглашаю вас исследовать этот огромный мир прагматичного проектирования на уровне типов!

1.3. РЕЗЮМЕ

- Обычный код (в статически типизированных языках) неизбежно является смесью решений уровня значений и уровня типов.
- Код уровня типов априори более сложен, чем код уровня значений. Правда, в отдельных случаях это не так.
- Программирование на типах должно быть оправдано тем, что привносит в код новые смыслы.
- Принципы проектирования помогают измерять сложность решения и избегать плохих технических решений.
- Диаграммы типизированных форм помогают проектировать типы.
- Типобезопасным называется код, имеющий хорошее с точки зрения типов представление, которое трудно использовать неправильно.
- `Newtype`-обертки полезны, когда требуется сделать код более типобезопасным.
- Программирование на типах начинается с освоения обобщенных типов, классов типов и типов-литералов на базовом уровне.
- `Haskell` допускает типы-литералы, если активировано расширение языка `DataKinds`.
- Возможно преобразовать тип-литерал, существующий на этапе компиляции, в значение на этапе выполнения.
- Существуют типы-строки (`Symbol`), натуральные типы-числа (`Nat`) и другие роды типов.
- Преобразовать значение времени выполнения в тип очень трудно, для этого нужна более мощная система типов, чем та, что имеется в `Haskell`.