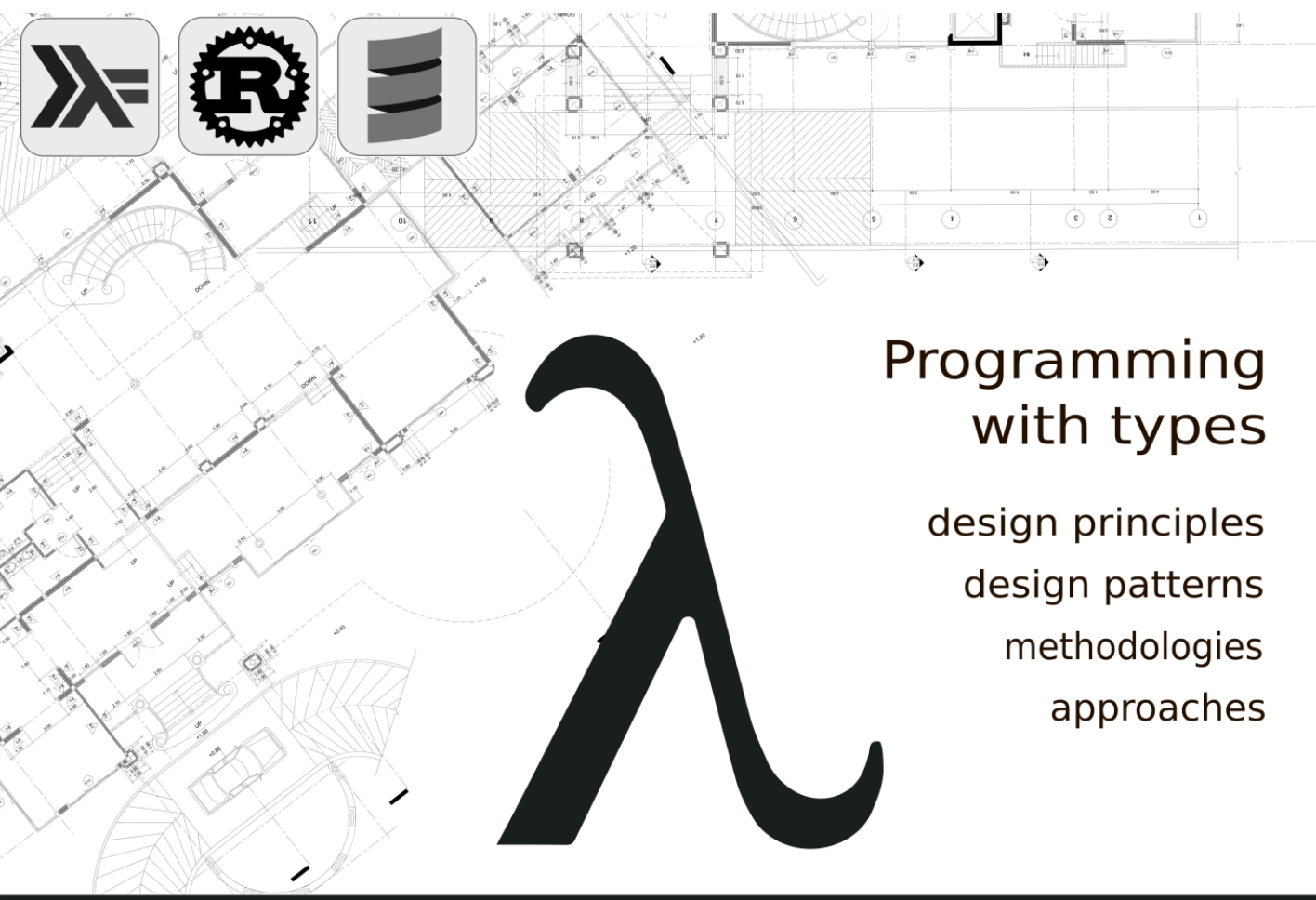

Programming with types

design principles
design patterns
methodologies
approaches

Pragmatic Type-Level Design

Alexander Granin

Pragmatic Type-Level Design

v.1.2.0-Sample Chapter

Alexander Granin
graninas@gmail.com

Author's Foreword

Hello dear friend,

Thank you for your interest in Pragmatic Type-Level Design! This book will be your guide through the fascinating world of type-level programming, making it as accessible as a leisurely walk in the woods. The main narrative of the book is presented in Haskell, but it also includes a section called "Rosetta Stone," which offers translations of the same ideas and approaches into Rust and Scala 3.

As you may know, I'm also the author of *Functional Design and Architecture: Examples in Haskell* (Manning Publications, 2024), a deep and novel exploration of applied, practical programming with Haskell and other functional languages. In *FDaA*, I cover high-level ideas, design principles, and best practices for building robust applications, offering a comprehensive source of knowledge about software engineering. The goal is to equip readers with everything they need to design and implement applications with functional approaches effectively.

However, as I worked on *FDaA*, it became clear that the world of type-level concepts lacks established development practices. This realization inspired me to write *Pragmatic Type-Level Design*—to bridge this gap and approach the subject in a way that sets it apart from other resources.

Type-level programming has been around for quite some time, with many languages offering rich type systems. It's a domain many cherish, particularly in Haskell, where types are among the language's most compelling features. Haskell developers often gravitate toward the mathematical foundations of types, such as Category Theory and Type Theory. However, as I sought to deepen my understanding of types, I often struggled,

frustrated, and confused. The academic emphasis on mathematics didn't provide the answers I was seeking.

As a pragmatist, I needed a clear, practical framework for software design with types, covering everything from design principles to implementation details. However, most resources failed to answer my questions. They were either overly academic or too fragmented to provide a coherent perspective. While type-level programming is inherently challenging, it's even more so in Haskell due to the disjointed and complex nature of the available materials. These resources often felt like folklore or scattered wisdom rather than structured, practice-oriented knowledge.

I struggled with those overly difficult concepts, not because I couldn't understand them, but because the traditional teaching methods didn't align with my perspective on types. If you feel the same, this book is for you. It's written in a fun, engaging style, filled with practical examples, and free from unnecessary mathematical complexity. You'll find it immediately useful for solving real-world business problems.

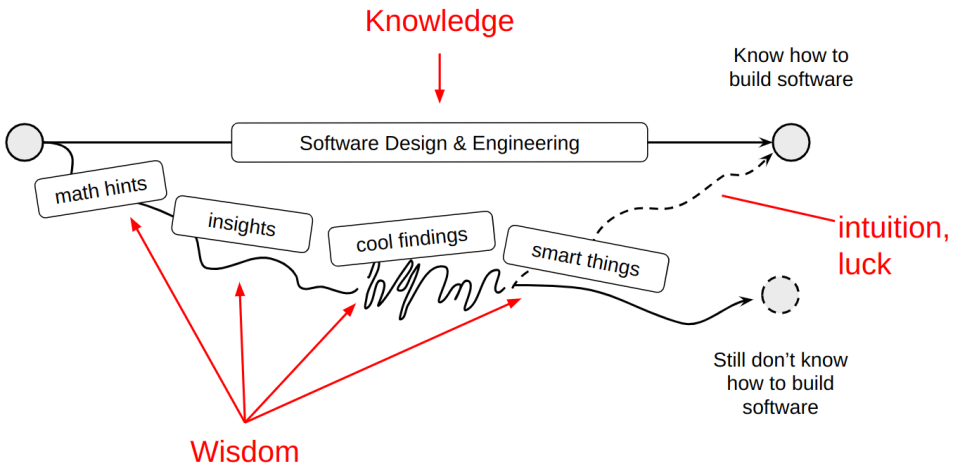

Figure: Wisdom versus Knowledge

With my book, type-level programming has become a practical and organized discipline, free from unnecessary academic abstractions and focused on real-world applications. Alongside *Functional Declarative Design* (FDD) —a methodology I introduced in *Functional Design and Architecture*—we now also have the *Pragmatic Type-Level Design* (PTLD) methodology.

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

> **LINK** Code and materials for *Pragmatic Type-Level Design*
> https://github.com/graninas/Pragmatic-Type-Level-Design

I hope you'll enjoy the book.

Best wishes,

Alexander Granin

Table of Contents

Part I

The Basics of Type-Level Software Design

Chapter 1

Emergent type-level design

This chapter covers

- How to program with types and values
- What is a *pragmatic type-level design* methodology
- Basics of type-level programming

What is type-level design? Before answering this question, I'll make a sudden parallel between type-level programming and cellular automata. This is not only a distant associative comparison; the philosophical relation between the two worlds is much deeper. We'll see this in action while working on a type-level cellular automata application throughout the book.

Do you like cellular automata as I do? What a wonderful invention it is – the glorious Game of Life by John Conway! The impressive depth of this cellular automaton emerges from an elementary set of rules, which boggles the imagination when you see these worlds for the first time. Just two rules on how a square field of neighboring cells evolves step-by-step – and you're getting a complex universe full of strange beasts that travel, oscillate, collide, merge, diverge, are born, and die.

Game of Life is Turing-complete, so it won't be a mistake to call it an esoteric 2D graphical programming language. Coding in this language has a particular charm. On the infinite quadratic

grid, we turn cells on and off, thus painting the program intended to mind some cellular business. The rules, therefore, act like a computer that recognizes this monochrome program and transforms the input world into the output world, one evolution after another. We prepare the pattern, run the world, and, after a certain number of iterations, get the result that hopefully has some desired properties.

The iterative evolution of the world is deterministic. Similar to pure functional programming, starting with identical cell patterns yields identical outcomes. However, unlike programming, predicting the result from the initial pattern is significantly more challenging, even after several iterations. Due to this, Game of Life programs are susceptible to bugs. Any mistake, misplaced cell, and our carefully arranged world blows up to a state we didn't expect. What was a recognizable pattern becomes chaos full of dancing cellular demons. What was once meaningful information corrupts and decays into elementary pieces, or sometimes into nothingness.

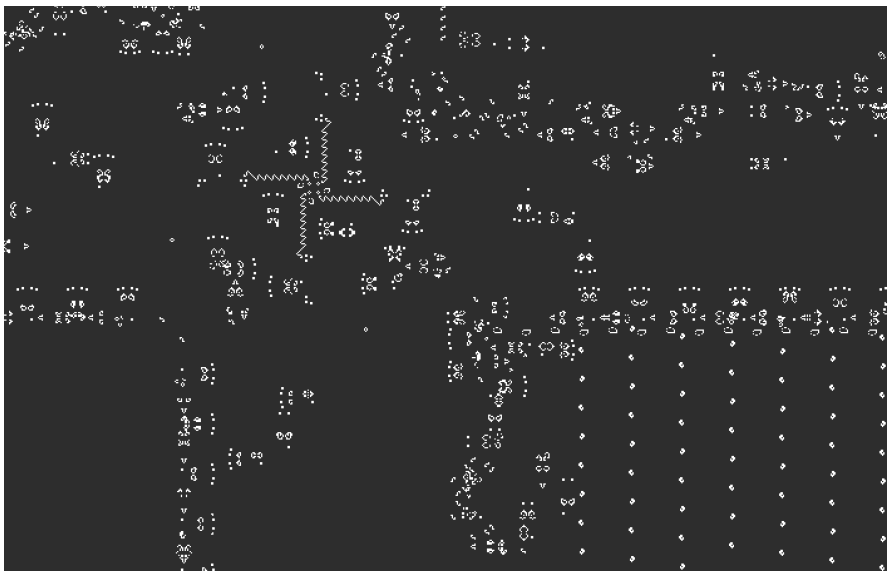

Figure 1.1 Metapixel: Game of Life built in Game of Life (fragment)

This is why programming such a system is quite a challenging task. Knowing both the rules of Game of Life and the fact that it's Turing-complete doesn't help. GoL programming is still a trial-and-error process, and many best practices have been developed to ease this process. Cellular design patterns, ready solutions, configurable libraries of mechanisms, debugging techniques – everything to deal with the complexity that naturally emerges from the simple base premises.

Sounds familiar, right? Indeed, everything said can be applied to type-level programming, too. Both started as computer science topics. Both promise a lot of joy to play with. Like a cellular program, type-level one starts with small, simple concepts but tends to blow into a tricky, complicated contraption that is difficult to observe and predict in behavior. Similar to Game of Life, a minor yet incorrect change in the type-level code can lead to unintended consequences.

Being a sharp tool, type-level programming can easily strike back and will certainly do so if treated naively. In the face of complexity, the evolution of GoL programming has led to having best practices, and so should type-level programming. Building big yet manageable programs with massive involvement of types should be done consciously, with good, reasonable, and controllable approaches. In other words, we need type-level programming to be engineering, not just a form of art or blind hacking. Hence, the book. With its pragmatic ideas, many software worlds will be saved from a painful and expensive death by a thousand emergent complexities.

1.1 Approaching the type-level design

Emergent systems are all the same. Given a small set of simple pieces and a rule for combining them, you can construct a highly complex mechanism that serves specific needs.

The same is true for biological life with its DNA foundations. Four nucleotides constitute every double spiral, but the number

of lifeforms emerging from this encoding boggles our imagination.

The same is Game of Life. A rule for a 2D grid of on-off cells that every child can understand leads to worlds we've never seen before.

The same is a type-level design.

> **DEFINITION** ***Type-level design***: software design focusing on applying various type-level features and tricks to solve the usual programming problems at compile time. Type-level design refers to the preference for type-level solutions over value-level ones, as well as for generic code over specific code, and compiler-time evaluation over runtime evaluation. Type-level design results in mechanisms and business logic expressed with type-level code.
>
> **DEFINITION** ***Type-level feature***: any language feature related to types and type transformations.
>
> **DEFINITION** ***Value-level design***: the opposite of type-level design. Programming tasks and business logic are expressed with values and functions. Types primarily describe the data but serve no additional purpose. Value-level design is also known as data-oriented programming.
>
> **TIP** Real projects in statically typed languages are always a mix of type-level and value-level solutions.

Type systems in such languages as Haskell and Scala are rich and full of sophisticated ideas, but even a subset of basic features may help us in our day-to-day needs. You'll be surprised by how much you can achieve without abusing every shiny thing in your type system and how much useful code it's possible to build. This means that while being a broad topic, type-level design is also an emergent system: a small number of carefully chosen ideas gives you a lot of possibilities.

But before submerging ourselves into this wild ocean, we should establish the ground and discuss what programming with types and values is like.

1.1.1 Types and values

If we were asked to create a program with a toy functionality, our engineer's reasoning would not go much beyond simple decisions. A minimal Game of Life application would be a good example of this. We could get the following requirements:

- Game of Life is a console application.
- The application accepts the number of iterations to do with the world.
- The application accepts the input file of the initial world in a predefined text format with a maximum size of 100 kilobytes.
- The application accepts the output file name for the final world in a predefined text format.
- The application performs the evaluations of the world and prints the results into the output file.

These are all functional and non-functional requirements for now. Nothing here is said about performance, and there is no requirement to track the intermediate worlds during the evolution. We can merely store the current world in an array-like or map-like data structure in memory. This might be suboptimal because the worlds often grow indefinitely in size, but at least this technical decision enables us to write the application quickly.

During the usual domain modeling, we'll end up with a typical code that relies on some data types to store the worlds. Nothing beyond that, nothing fancy or generic, just some types to denote some runtime values our program should operate with. The following `Board` maps indices to the states of cells (see figure 1.2 for a Typed Forms diagram):

```
type Coords = (Int, Int)              #A
```

```
data Cell = Alive | Dead                #B
type Board = Map Coords Cell            #C

#A Coordinates
#B Possible cell states
#C Dictionary of cells (2D field)
```

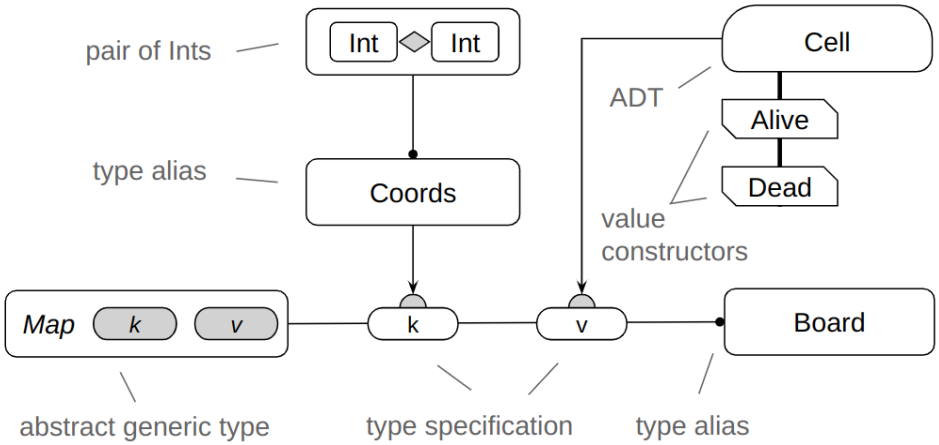

Figure 1.2 Typed Forms diagram for Board

> **TIP** A complete guide on Typed Forms diagrams is available in *Appendix A: Typed Forms diagrams.*

What can be simpler than that? Just a two-dimensional key that points to a specific cell on the board. Every cell here is encoded directly, but enumerating dead cells is unnecessary. The following glider pattern contains five living cells:

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

The shape of this glider is presented in figure 1.3:

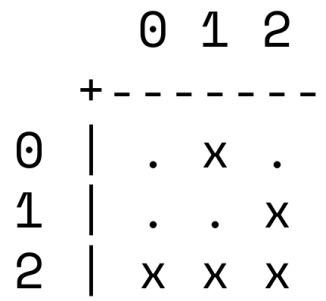

Figure 1.3 Glider

The initial application code may look like the `main` function, having all the read-write file operations and some pre-hardcoded constants. This one iterates the world five times:

```
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 Load the initial world
#B Do five steps
#C Save the final world
#D Functions that do actual stuff
```

Types here serve a descriptive purpose. The `loadFromFile` function returns a value of `Board`, `iterateWorld`, then uses the value to get the next value, the fifth generation of the world. The type `Board` itself doesn't do anything except make our values better readable and understandable. We could stick with the bare `Map` type instead and have the same functionality because the types are just synonyms:

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

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

Nothing has changed here, so we can conclude that the `Board` type identifier doesn't play that much of a role in our code. We only have it because it's more convenient; otherwise, it provides no extra meaning.

Some new meanings could be introduced if we start wrapping this type synonym into a **`newtype`** (see figure 1.4 for a Typed Forms diagram):

```
newtype GoL = GoL Board

iterateGoL :: Int -> GoL -> GoL
```

> **TIP** Other languages have similar features to Haskell's **newtype**. See *Part IV: Rosetta Stone* for more info.

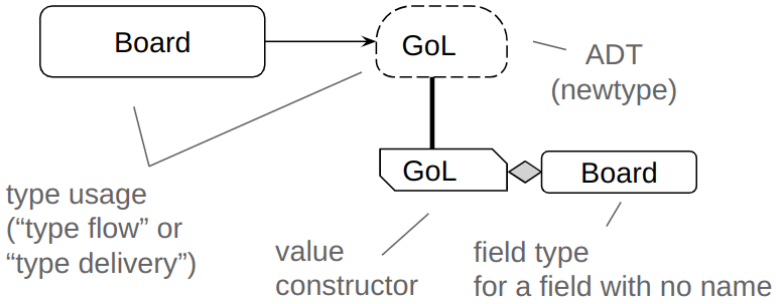

Figure 1.4 Typed Forms diagram for GoL

Adding one more newtype wrapper for a different automaton, Seeds:

```
newtype Seeds = Seeds Board
```

allows us to distinguish the two types (like `GoL` and `Seeds`) at compile-time, and they cannot be mutually replaceable, even if the stored boards are identical.

A common example is a person's first and last name. It should be a character string type, but string types are everywhere in projects and mean many things. So we do the newtype encapsulation:

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

It's now difficult to confuse the two with `FilePath`, which happened to be of type `String` as well:

```
type FilePath = String

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

lastFileName :: FilePath
lastFileName = "last_file_name.txt"

main = writeFile lastName "McCarthy"     -- compile-time error
```

The trick helps to save some worlds from annihilation at the cost of a little inconvenience and boilerplate. We have to unpack the newtype to access its contents:

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

main = writeLastName lastFileName lastName
```

#A Unpacking by pattern matching

Passing `FirstName` into this `writeLastName` function is impossible, so this code is too specific and rigid without additional mechanisms.

This feature alone doesn't let us climb much higher from the usual code with basic types and values. Our reasoning still revolves around transforming values in the runtime and doesn't touch on how to modify the values during compile time. We don't have any transitions of types to values and back, and we

don’t have any types transformed to any other types – the kind of operations that constitute the core of type-level design. In our case, types describe, values labor, and nothing beyond that.

You can do everything with this good old style of programming. There is no need to bring extra meanings into the types; just call functions, process data, express your domain in ADTs, and enjoy the full power of regular functional programming. If you need best practices for this path, follow my methodology, *functional declarative design* (FDD). It covers this programming style well and provides many useful ideas on making your functional code good. Consider learning the methodology in my *Functional Design and Architecture* book:

> **LINK** Alexander Granin, *Functional Design and Architecture*
> https://www.manning.com/books/functional-design-and-architecture

Alternatively, you can read my introductory article:

> **LINK** Alexander Granin, *Functional Declarative Design: A Comprehensive Methodology for Statically-Typed Functional Programming Languages*
> https://github.com/graninas/functional-declarative-design-methodology

Going type-level means doing type transformations and making code even more abstract. This often produces complexity and can make the code less obvious. Type-level features in all languages bring a significant syntax and semantics overhead. So, refraining from value-level programming and shifting to type-level programming should be properly justified. There should be a convincing reason for doing this—a sensible goal that reflects real business needs, not imaginary ones. In other words, adopting type-level programming should be pragmatic; otherwise, we risk introducing unnecessary complexity without gaining any meaningful benefits.

1.1.2 Pragmatic type-level design methodology

Resources on types exist in numbers. Many are trying to answer the question, "Why is the type-level stuff cool?". Sometimes, they try to teach you math instead of writing useful software. The authors are keen to show you the beauty of type-level tricks and share the related enthusiasm around type-level mind games.

Pragmatism comes from a different side. It addresses these questions: "How to apply type-level design for real tasks?" "How can we achieve business goals without exceeding the complexity budget?" "Why is this type-level solution better than a value-level one?". Here are my answers when the type-level design looks good:

- better ways to express critical business domains;
- lesser bugs and greater guarantees for code correctness;
- lesser need for extensive testing;
- more extensible code;
- more reusable generic code;
- pre-calculating things at compile time;
- code generation;
- performance optimizations and zero-cost abstractions.

The reasonable approach should be about achieving some meaningful, considerably better result when it's heavily typed. The complexity of the type-level code may still strike back eventually, so the possible refactorings should not lead to the whole application rewrite. Such solutions are often difficult to implement and understand, and they tend to become cryptic even for the author after a while. Sometimes, you feel like the protagonist of the "Flowers for Algernon" novel when his cognitive abilities collapse, and he becomes unable to understand his writings.

My methodology, *pragmatic type-level design* (PTLD), addresses these issues. It acknowledges the need to control accidental complexity and weigh all the cons and pros of a type-level solution against others.

> **DEFINITION** ***Essential complexity***: inherent, inseparable complexity of a business domain that reflects its essence. Programming a domain and avoiding bringing essential complexity into the project is impossible.

> **DEFINITION** ***Accidental (incidental) complexity***: the complexity of tools and design decisions used to implement a business domain and make it work. Solutions and approaches vary in their accidental complexity in the given contexts. Some solutions bring more accidental complexity; some get less under similar circumstances. The more accidental complexity in the project, the more expensive it is to evolve it, and the bigger the risk of it failing technically.

Accidental complexity is the main enemy of Software Design, including Type-Level Design. Our software architects' meta-goal is managing complexity and taming it so it does not ruin the project in the long term. Accidental complexity tends to grow if we don’t do anything to keep it low. Arguably, the more type-level features are used, the steeper the growth of accidental complexity in the project with time. Figure 1.5 brings this point into a plot:

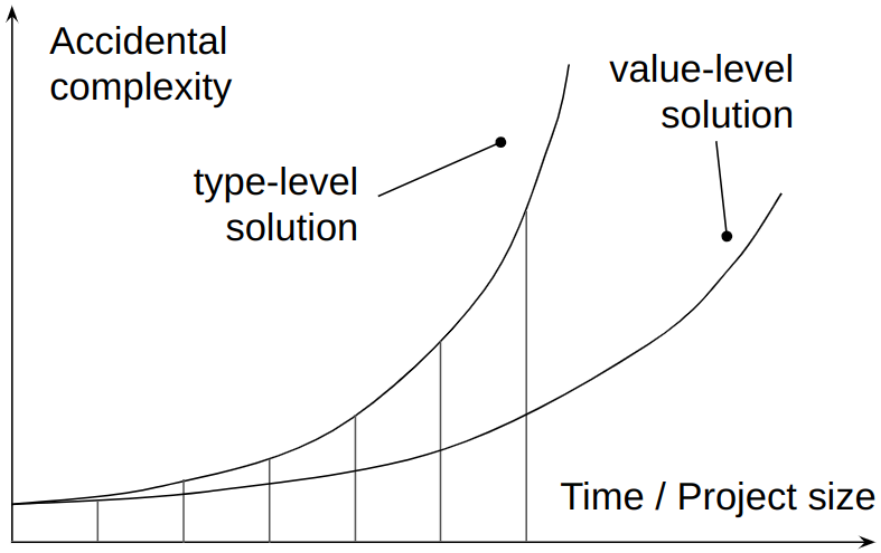

Figure 1.5 Accidental complexity of type-level and value-level solutions

We should be careful in our steps because type-level programming carries a carriage of risks attached to it:

- *Overengineering*. Chosen design solutions have an unnecessarily high accidental complexity that will greatly damage the project's budgets and long-term health.
- *Overcomplicating the code* means making it difficult to understand and modify by choosing bad names and introducing too abstract, too generic solutions. It often means dissolving the business domain in the auxiliary mechanisms, resulting in a complete inability to recognize domain notions and behavior in the code.
- *Invalid design decisions*. Type-level programming offers many ways to do the same thing. Inappropriate solutions can suddenly become an obstacle to further development of the code. They may result in completely rewriting significant parts of the project or abusing hacks that could worsen the situation.

- *Pleasuring one's own narcissism*. Type-level programming, generic programming, and metaprogramming may be very tempting for people who focus on their intellectual pleasure and forget about why they are hired in the first place. It's easy to substitute business goals with the desire to show off smartness and the very ability to mangle with high-level, complex ideas. Still, we, software engineers, should always be honest and technically correct when justifying our decisions.

To avoid these risks and stay rational, I offer the following general principles within the PTLD methodology:

> **PRINCIPLE I** ***Occam's razor***. It's better to stick to a limited set of features to cover 95% of cases and not introduce extra features without a real need.
>
> **PRINCIPLE II** ***No perfectionism***. No need to make 100% of the code perfect, correct, and beautiful. If covering all the cases with a type-level code promises to explode in complexity, it's better to try other solutions instead.
>
> **PRINCIPLE III** ***Dumb but uniform***. A single, even dumb, methodology of doing everything uniformly is more valuable than a brand-new, shiny solution for every next problem.
>
> **PRINCIPLE IV** ***Pragmatism***. Beauty and math correctness are of less value than an imperfect but working system. Shortcuts and tradeoffs can be traded for less accidental complexity. Design should always pursue real business goals.

Pragmatic type-level design, therefore, is a lighthouse in the darkness, a map of treasures that will guide you through the wild and untamed world full of scary beasts and dangerous temptations.

Figure 1.6 compares the three methodologies: *object-oriented design* (OOD), *functional declarative design*, and *pragmatic type-level design*.

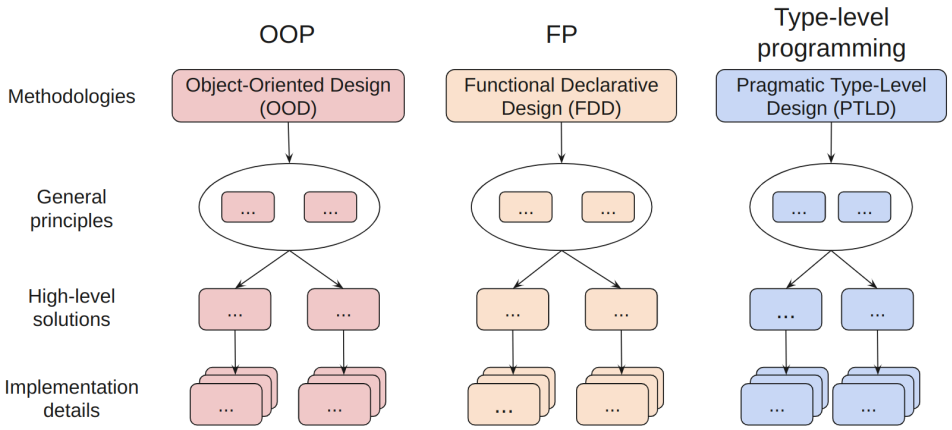

Figure 1.6 Three Software Design Methodologies

We have already discussed some general principles and still have more to discover. All the design principles we know in OOD and FDD have a good chance of being suitable for PTLD. We'll meet many of them soon: SOLID principles, low coupling and high cohesion, divide and conquer, and many others. This diagram also tells us that high-level technical solutions follow directly from these principles, and only then do the implementation details. Figure 1.7 highlights the principles and methodologies:

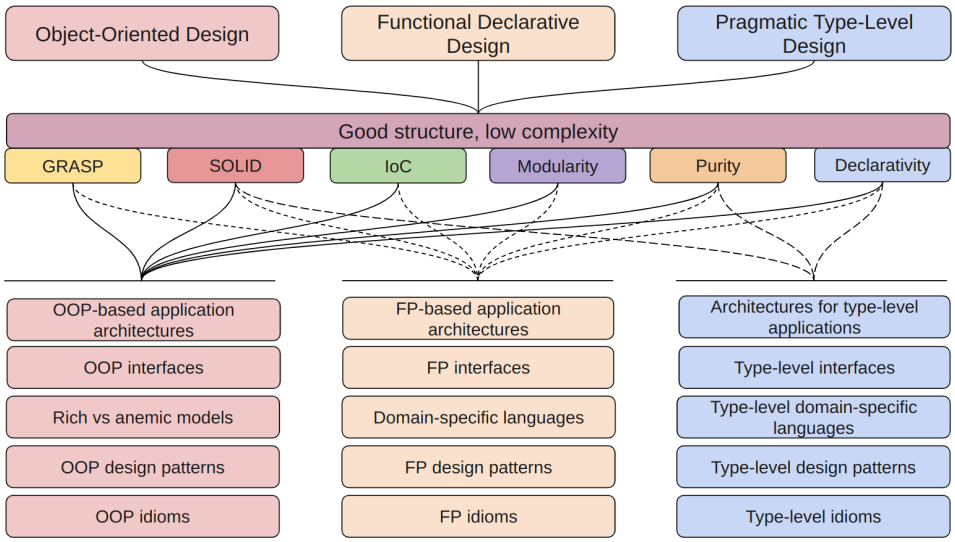

Figure 1.7 Design principles and ideas to keep the complexity low

I especially look forward to discussing type-level domain-specific languages as my favorite way to express business domains and simultaneously beat the accidental complexity. Consider this, with no explanations; it's a full-fledged type, not a runtime value:

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

It's all Haskell's types describing some logic for a cellular automaton on the pure type level. We'll come here eventually. However, before jumping this high, we must learn some basic type-level tricks and ideas.

1.1.3 Typed Forms diagrams

Learning type-level stuff may be quite challenging. I know this well. It's really easy to get lost in all those definitions and interconnections, especially when the syntax of the host

language is awkward. In this sense, Haskell is certainly not C++, but sometimes, the sophistication of the syntax and semantics exceeds all reasonable limits.

I've developed a special visual language of structured diagrams for my PTLD methodology. I call it Typed Forms, and I'm sure it will serve two good goals for us: making the explanations clearer and making the book aesthetically appealing. I'll teach you this diagram language when the time comes. You can also consult the full description in *Appendix A: Typed Forms diagrams*.

This is an example of such a diagram:

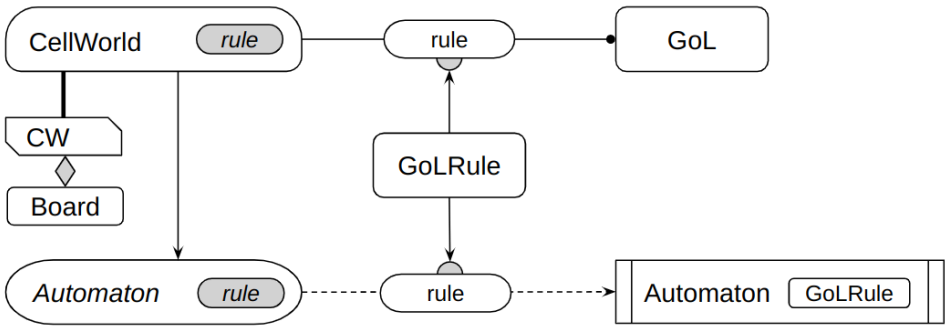

Figure 1.8 Typed Forms diagram

It looks cool, do you agree?

1.2 The foreshadowing of type-level design

Good old Haskell 2010 introduces a lot of type-level features. Type definitions, type classes, type variables, constraints, kinds, phantom types, and algebraic data types. With dozens of modern GHC extensions, it becomes more powerful. Type-level literals, type families, multi-parameter type classes, generalized algebraic data types, different options for type-level polymorphism, et cetera, et cetera – it's too long to enumerate all these tools, what to say about the explosion of all possible ways to combine them. This is why I believe the field of type-level programming is a pure art, yet everyone does it differently. We will turn this art into

engineering so it can be applied to various tasks uniformly. But we have to learn the fundamentals first.

I will demonstrate two important features: type classes and type-level literals, which are at the heart of type-level design. We'll introduce many other type-level features as needed.

1.2.1 Basics of generics

Let's return to the Game of Life application. I have some news: the functional requirements have suddenly changed, as often happens in the real world. We're now mandated to program the code that supports all possible Life-like cellular automata, not only the one John Conway has invented and described.

Remember the `GoL` type that is a wrapper over the `Board` type? Here we have three of them for various rules:

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

#A Different Life-like rules: Game of Life, Seeds, Replicator

All three are Life-like rules, which means they all work on a 2D grid with 2-state cells and use the same 8-cell area of neighborhood squares. The rules of how cells turn on and off differ, and we normally don't want the `Seeds` rule accidentally applied to the `Replicator` world. With the newtypes, we prevent this mismatch, although we have to have several separate step functions for each:

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

We've obtained some safety by appending the extra knowledge to the initial `Board` type. This slight improvement was nice and easy, and the compiler doesn't care about one more newtype or one less. But we are not machines. There are 2^18 = 262144 possible Life-like rules, and only a tiny fraction of these are

investigated. Imagine we have to add more and more wrappers when more cool automata emerge from the community of lifers:

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

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

#A Three more Life-like rules: Diamoeba, HighLife, DayAndNight

Wow, much safety, so boilerplate. Iteration functions will breed, too:

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

On a closer look, however, these functions are all identical if we presume they use a function for a single step of a particular automaton:

```
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 Game of Life step
#B Game of Life rule here
#C Taking the sixth item from the list of iterated worlds
#D Dropping the first n items from the list of iterated worlds
#E Infinite list of iterated worlds; invocation of the step function

The `iterateReplicator` function will have the same body, up to the identifiers. This is where generics come out. We can move all this logic into an abstract function that accepts a step of any cellular automaton function and does the same: iterates the given world with the given rule `n` times:

```
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"
```

It really doesn't matter what the `ca` type is as long as the `step` function is provided. There is no need to know about the internal structure of the factual `ca` type because this knowledge is conveniently encapsulated and utilized within the `step` function. We've even achieved some type safety here. It's impossible to apply the step function to the wrong world:

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

> iterateWorld golStep   5 glider'      -- OK
> iterateWorld seedsStep 5 glider'      -- compilation error
```

> **DEFINITION** ***Type safety***: inability to perform wrong type conversions, to construct incorrect types, or to combine types and values that are not supposed to be combined.

This works because all four `ca` type variables (`ca` stands for "cellular automaton") should be the same type within a single invocation of `iterateWorld`.

> **TIP** Other languages have generics as well. See *Part IV: Rosetta Stone* for more info.

What should we do with other useful functions that need to know about the internals? This is the saving function:

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

We certainly don't want to produce many separate methods like `saveGoLToFile`, `saveSeedsToFile`, and so on, and at the same time, we can't generalize it yet:

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

We know that all the newtypes wrap the same `Board` type, but this function has no idea about that, and it can't see through the `ca` type variable here. The value of this unknown `ca` type is opaque to the function, and there are no hints on what this is if fully specified. This means our generalization is limited, so we have to find another way.

1.2.2 Basics of type classes

Type classes will help us to generalize automata functions in a more rigorous way. For now, we've observed the following pieces of each automaton:

- a type that distinguishes one automaton from another;
- a specific step function that performs the corresponding rule.

We've also found that for the `saveToFile` function to work, we need to unwrap the `Board` value from the automaton type. This could be done pretty straightforward:

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

And then:

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

seedsWorld :: Seeds
seedsWorld = Seeds glider

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

However, while completely valid, I would argue that this design forces us to stay on the value level and carry out all those helper functions. But now we're ready to introduce a type class interface for any possible Life-like cellular automaton. It will only have three methods:

Listing 1.1 Type class interface for cellular automata

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

We're implementing it differently for the newtypes `GoL`, `Seeds`, `Replicator`, and so on. This is perfectly sensible: the types are distinct and unique, so there can be a unique instance of this type class for each. If we tried to instantiate the type class with `Board` several times, it wouldn't work.

> **TIP** Instances of type classes will clash as long as they are made for the same type and are imported into the same module. Type class instantiation has other hidden pitfalls, so consider learning about them from other sources.
>
> **LINK** Vitaly Bragilevsky, *Haskell in Depth*
> https://www.manning.com/books/haskell-in-depth
>
> **TIP** C++, Scala, PureScript, and some other languages have mechanisms similar to type classes. See *Part IV Rosetta Stone* for more info.

Listing 1.2 Two separate implementations of the type class

```
-- 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 Implementation for GameOfLife
#B Implementation for Seeds

A Typed Forms diagram is presented in figure 1.9:

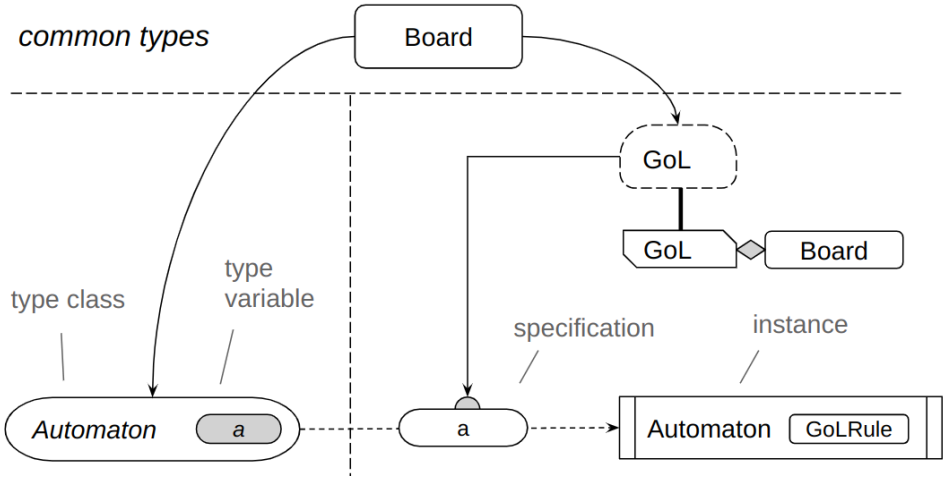

Figure 1.9 Type class and instance

Now, given these instances, our generic functions become more convenient to use because we don't need to pass helper functions anymore. Everything needed can be requested from the specific instance of this `Automaton` type class, and our right to do so is written in the type declarations:

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

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

All these methods know that the actual type that comes instead of `ca` must have `step`, `wrap`, and `unwrap` implemented, so it's legit to call these functions when needed:

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

#A Method 'wrap' from the type class

Finally, we can explicitly specify what implementation we're dealing with. In the following snippet, the compiler will deduce that those functions should come from the instance related to the `GoL` type:

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

let gol2 = iterateWorld 5 gol1

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

#A Explicit type specification
#B Generic function call

This is how we got some extensibility with type classes. It's now possible to add more cellular automata without updating the generic functions. The new instance doesn't even have to be a newtype around `Board`. For example, I could represent my world with an associated list instead of the dictionary:

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

Additionally, I've put another field into this ADT to denote the name of the world. This field will not create obstacles to the `Automaton` mechanism, although it doesn't help that much. We

should be prepared that the name may get lost or ignored eventually:

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

#A Default empty name is provided
#B The name is lost

Otherwise, the design seems to be valid.

Or does it?

Not quite. It has some more serious flaws and limitations and even violates a pair of design principles, which is not good. Let's talk about that.

1.2.3 Design principles

Look at this tiny, innocent type class again. How can this sweet child violate something?

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

In fact, it doesn't respect the two SOLID principles: the *single responsibility principle* (SRP) and the *interface segregation principle* (ISP).

SRP says that an entity should have only one responsibility. `Automaton` has two: progressing the world with `step` and accessing the `Board` value with `wrap` and `unwrap`. What about these two? Having them in the type class violates the ISP principle as they are domain-unrelated. The principle encourages us to group responsibilities through separate interfaces and not mix different levels of abstraction, which we did here.

Design principles such as SOLID highlight design smells and allow us to argue about issues reasonably. In contrast to a common misbelief (that persists across the development community), SOLID principles, first coined by Robert Martin, are not theory; they are practical and useful.

We do not just hack the code in our design process because this would be blind and anti-engineering. When we design, we are guided by high-level arguments to see if our current idea is good enough. SOLID principles, and some others, such as "low coupling, high cohesion," being treated correctly, improve the overall quality of the code.

- *Single responsibility principle (SRP)*. An entity should address only one responsibility at a time.
- *Open-closed principle (OCP)*. The system should have stable interfaces, and adding more implementations or new behavior should not affect the agreed-upon contracts of behavior (open for extension, closed for modification).
- *Liskov substitution principle (LSP)*. Multiple implementations are interchangeable on the fly without breaking the client code.
- *Interface segregation principle (ISP)*. Responsibilities should be spread across focused and coherent interfaces.
- *Dependency inversion principle (DIP)* states that business logic should depend on interfaces only, not on the actual implementation provided later in the interpreting process.
- *Low coupling, high cohesion*. An entity should depend on the bare minimum of external, unrelated entities (low coupling). It should contain only what it needs and should not have extra unnatural responsibilities (high cohesion). This principle comes from another set of principles known as GRASP (General Responsibility Assignment Software Patterns).

These principles were initially cast for *object-oriented design*, but they are universal. In my FDaA book, I've demonstrated

their validity for functional programming and how this makes the *functional declarative design* methodology rich and powerful. Certainly, *pragmatic type-level design* wouldn't be pragmatic and wouldn't be a design without these principles, and you'll see how to apply them to the level of types.

Let's revisit the newtypes approach because it's suboptimal and brings too much boilerplate into the code. We'll fix this by introducing a new world type and utilizing another interesting feature: type-level literals. The new design will be good enough to further improve the `Automaton` mechanism in the next chapter.

1.2.4 Basics of type-level literals

There is a good property of the previous design: we could have a world of any internal structure. For example, it could be a dictionary-like type `Board`:

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

Or a real 2D array:

```
import Data.Vector

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

#A 2D array

This feature has nothing bad, but it's mostly a regular design and nothing about a type-level one. We'll sacrifice the possibility of different world structures for different automata for a while. We'll also avoid sparking newtypes for every possible world we need. There will be only one type predefined for all the automata, similar to this:

```
newtype CellWorld = CW Board
```

If we try to make the actual automata by just typing synonyms, we'll encounter a problem.

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

These two types are literally the same; they're just two identifiers for the same type. Going ahead, this won't work with the `Automaton` type class as we can't instantiate it twice. We should make them very distinct. One way to do it is parametrization on the type level, for example, with type-level strings. So, there will be a parametrizable `CellWorld` template and specific types for automata that differ on the type level.

First, this is how specific worlds look now:

```
{-# LANGUAGE DataKinds #-}

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

type GoL   = CellWorld GoLRule
type Seeds = CellWorld SeedsRule

#A DataKinds used here
```

> **TIP** Type-level strings are types, and they are distinct if the content differs.

> **TIP** Type-level strings require the `DataKinds` GHC extension to be enabled. With this extension, the compiler treats bare strings as types in the corresponding places.

> **TIP** Type-level strings and other type-level literals are not unique to Haskell. See *Part IV Rosetta Stone* for more info.

Second, we should redesign the `CellWorld` type. Just giving it one arbitrary type parameter would work partially:

```
newtype CellWorld arbitraryType = CW Board

type SomeIntWorld = CellWorld Int
```

`arbitraryType` here can only hold regular types, such as `Int`, `Char`, and others. It will not accept `"Game of Life"` and `"Seeds"` from the code above. These types are not arbitrary; they are type-level strings. In Haskell, this distinction is important and relates to the notion of *kind* (a type of type). Going ahead, `Int`, `Char`, and other regular types are of the `*` ("star") kind (more on this in the next chapters), while type-level strings are of the `Symbol` kind. Pseudocode:

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

#A Ordinary kind for regular types
#B Specific (unique) kind
```

A full description of the `CellWorld` presented above includes the omitted optional kind specification:

```
newtype CellWorld (arbitraryType :: *) = CW Board

type IntWorld  = CellWorld Int         -- Okay: Int is *
type SeedsRule = CellWorld "Seeds"     -- Error: "Seeds" is
Symbol, not *
```

We should reflect the `Symbol` kind in the definition of `CellWorld` for two reasons: it's a sort of documentation; the compiler must know this to make a proper type inference.

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

import GHC.TypeLits ( Symbol )
```

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

#A DataKinds + KindSignatures GHC extensions used here

The corresponding diagram is this:

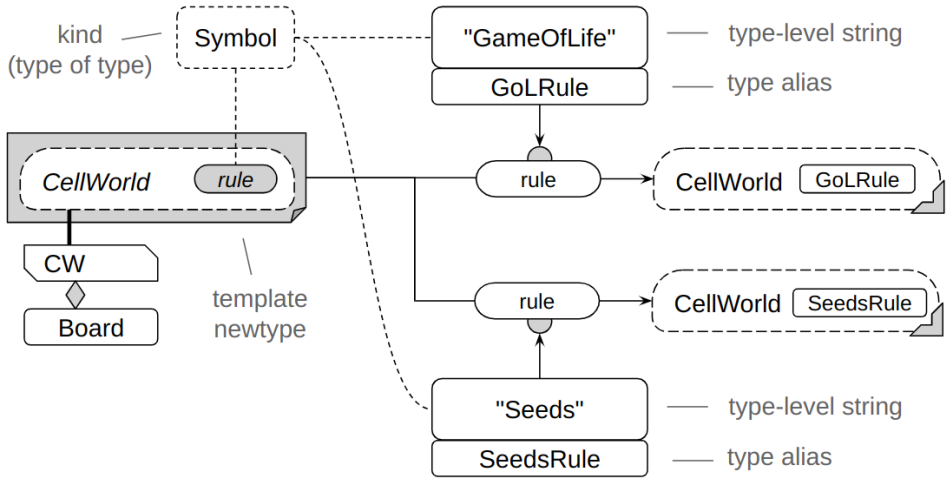

Figure 1.10 Typed Forms diagram for CellWorld

NOTE In the modern Haskell extensions, kinds were replaced and unified by the idea of "types of types." The special `Type` kind replaced the star symbol, and the distinction between types and kinds was eliminated. This made the whole type system even more complicated. To avoid being overwhelmed, we'll keep the old terminology for now.

NOTE Two extensions here, `DataKinds` and `KindSignatures`, are responsible for the `(rule :: Xyz)` specification form. Check this out:

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

```
import GHC.TypeLits (Symbol)

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

We have no choice but to follow the recipe without much explanation. We'll return to these extensions and declarations later.

NOTE The `CellWorld` type has the `rule` type parameter, but the `CW` value constructor doesn't utilize it. This makes the `rule` a phantom type; likewise, `iAmPhantom` here:

```
data SomeType iAmPhantom = NothingToSeeHere
```

Phantom types keep some extra information about other types employing free unused type parameters. As we'll see in further chapters, phantom types are important for the type-level design.

What else can we do with type-level strings? We're encoding the name of an automaton, and it would be nice if we could print it once we're given some world. This conversion from types to values in Haskell is simple, although it requires the help of several exotic creatures from the depths of the language. The zoo consists of:

- `KnownSymbol` type class from `GHC.TypeLits`;
- `stringVal` function that belongs to the `KnownSymbol` type class.

The following function does the conversion once provided with a `CellWorld` value:

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

automatonWorldName
  :: KnownSymbol rule             #A
```

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

#A Constraint requires `rule` to be of the Symbol kind
#B Denoting `rule` as Symbol is optional
```

Specifying the `Symbol` kind is optional because it is present in the definition of `KnownSymbol` and `CellWorld`. This is, however, allowed:

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

The `symbolVal` function sees no interest in the value itself but needs the actual string type carried with the `world` variable. With this `world` variable, we're proxying the needed type to `symbolVal`, so it could know what type-level string we want to convert:

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

The `automatonWorlName` function stares at the `CellWorld` type and sees that `rule` has the `Symbol` kind, so it's fully eligible for the `KnownSymbol` type class, and, therefore, for `symbolVal`.

Usage is simple as long as we have a world; for example, Game of Life:

```
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 also supports type-level natural numbers (from 0 to infinity). We could easily replace type-level strings with them and achieve a similar effect.

```
import GHC.TypeLits (Nat)

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

There is the `Nat` kind, the `KnownNat` type class, and the function that converts type-level numbers into value-level integers:

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

type FourtyTwo = (42 :: Nat)

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

Here, we only have the `FourtyTwo` type and nothing else. In the lack of a real value of this type, we're constructing `Proxy` from `Data.Proxy` that will help `natVal` to understand what we mean. `symbolVal` supports this trick as well:

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

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

Besides strings and numbers, there are type-level characters, type-level boolean values, type-level algebraic data types, and so on. We'll need all this stuff in our journey, piece by piece, step by step.

Thus, I invite you to explore this huge world of *pragmatic type-level design*!

1.3 Summary

- Regular code is usually a mix of value-level and type-level solutions.
- Type-level code has an a priori higher complexity than value-level code. This can be wrong for specific cases.
- Type-level programming should be properly justified and introduce new meanings into the code.
- Design principles help measure the solutions' complexity and avoid bad technical decisions.
- Typed Forms diagrams help design with types.
- Type-safe code is code that has a good type representation that is difficult to use incorrectly.
- Newtype wrappers are useful to make the code more type-safe.
- Type-level programming starts with understanding generics, type classes, and type-level literals on the basic level.
- Haskell allows for type-level literals with the `DataKinds` extension enabled.
- Converting a compile-time type-level literal to a runtime value is simple.
- There are type-level strings (`Symbol`), type-level natural numbers (`Nat`), and other type-level kinds of types.
- Converting a runtime value to a type is very difficult and requires a more powerful type system than Haskell has.

Conclusion

Complexity itself is an emergent phenomenon. The path to complexity often involves small, possibly simple pieces that, once connected, show properties that never existed in any part. Controlling complexity is recognizing an approach as overly complex and watching a system's relations and emergent properties.

Some complexity can be tolerated if justified or unavoidable. Still, it’s a good idea to try to localize it and hide behind boundaries so it does not leak into the most important parts of the system. However, the more complexity we tolerate, the harder it will hit us in the future, thus causing loss and problems (see figure X.1).

Pragmatic type-level programming is a sharp-edge walking activity, so we must be cautious with our chosen solutions. I hope this book has taught you to think like an engineer and provided you with the necessary tools for future endeavors. Certainly, there are even more topics to discuss:

- type-level calculations (arithmetics, physical units on the level of types);
- refined types (types that express bounds for type-level values);

- dependent types (types that can be built on the fly using value-level runtime data);
- linear types (types that can count usage occurrences and prevent excessive or insufficient use).

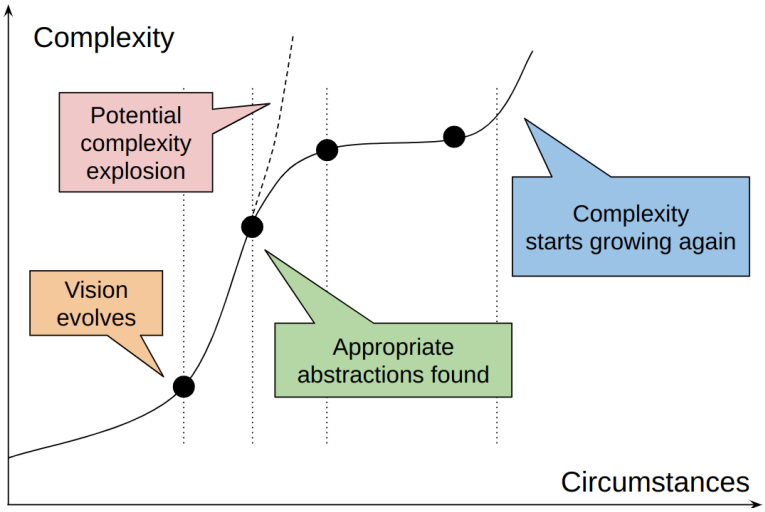

Figure X.1 The lifecycle of complexity

Thank you for reading the book!

And I would also be happy if you read my other book, *Functional Design and Architecture: Examples in Haskell*. In a sense, I wrote this one as a kind of promotional material. But it turned out to be a completely independent exploration. I hope you found it as interesting and useful as I did.

Part IV

Rosetta Stone

The Rosetta Stone is an ancient artifact. It is a stone stelle with three engraved writings. The same text is translated three times: Ancient Egyptian encoded hieroglyphic and Demotic (top and middle texts), and Ancient Greek (bottom texts). The text is a decree about the royal cult of Ptolemy V in 196 BC at Memphis. Discovering the stone has led to a breakthrough in deciphering the Ancient Egyptian language.

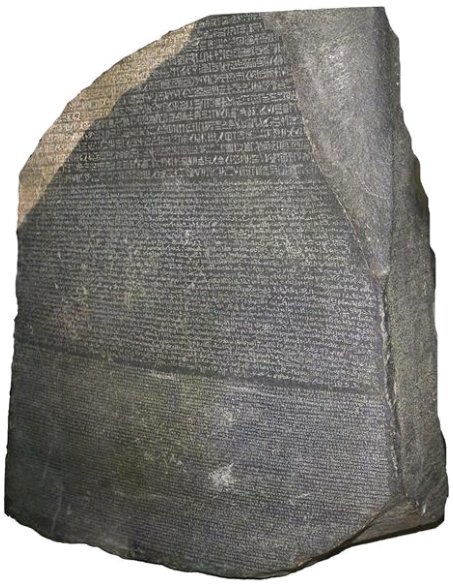

Figure RSC.1 Rosetta Stone

This part of the book, Rosetta Stone, is dedicated to translating the ideas of the books into other programming languages, when possible, and to the degree my expertise allows that. It is not reference material about features. It doesn't aim to highlight all the related details. It's a comparison of approaches and an analysis of the languages' abilities to fulfill the methodology initially described for Haskell.

Rosetta Stone Chapter 1

Rust

This material reiterates the approaches from the book for Rust. It is not intended to be a comprehensive documentation of Rust's features. For a detailed explanation, consider the official documentation. The material also relies on the notions and ideas from the main chapters, so in-depth explanations are avoided, assuming the book is read in full.

> **LINK** Rust Documentation
> https://doc.rust-lang.org/beta/

Rust's type system is generally much weaker than Haskell's (considering the GHC extensions), but it is enough for a proper type-level design. If Rust gets higher-kinded types, type equality, GADTs, and DataKinds in the future, some of the implementations from this section may be simplified.

Table RSC.1.1 Feature comparison for Haskell and Rust

Haskell	Rust
generics ●●●●●	generics ●●●●○
newtype feature	newtype idiom

●●●●●	●●●●○
ADTs ●●●●●	enums & structs ●●●●○
empty ADTs ●●●●●	empty enums empty structs ●●●●○
parameterized ADTs ●●●●●	parameterized ADTs ●●●●○
Proxy type: Data.Proxy ●●●●●	Proxy type: PhantomData ●●●●●
DataKinds and type-level ADTs ●●●●●	No DataKinds; custom type-level ADTs with empty parameterized ADTs ●●●○○
kinds ●●●●●	no native kinds; custom kinds ●●●○○
GADTs ●●●●●	no GADTs ○○○○○
multi-parameterized type classes ●●●●●	multi-parameterized traits ●●●●○
type families ●●●●○	associated types ●●●●○
type-level lists ●●●●●	no native support; custom type-level lists with macros ●●●○○
type-level strings ●●●●○	no native support; custom type-level strings with macros ●●●○○

<Continue with the full version…>

Rosetta Stone Chapter 2

Scala 3

This chapter contains the book's approaches applied to Scala 3 to the extent possible and reasonable. It is not a reference or documentation about Scala itself and doesn't discuss features in all their glory. Consider official documentation and external resources for more info.

> **LINK** Scala Documentation
> https://docs.scala-lang.org/

Scala 3 is a powerful language. Any task can be solved in multiple ways, which is also what makes it difficult: deciding on an approach may require a deep understanding—a luxury not everyone has. This chapter offers several approaches, but it is not exhaustive and may not be fully optimal. After all, I'm not an expert in Scala. Also, from the folklore, I know that Scala developers love playing with types. They eat types for breakfast and can dissertate on the limits they pushed the language beyond. Take Shapeless and Cats – these two worlds alone can show how far they've gone. They will certainly find my writings like a Stone Age rock painting. However, my goal is not to impress elite Scala developers; it is rather to help those mere developers like me who want

something less advanced and yet more practical in their day-to-day work.

Table RSC.2.1 Feature comparison for Haskell and Scala 3

Haskell	Scala 3
generics ●●●●●	generics ●●●●●
newtype feature ●●●●●	newtype idiom ●●●●○
ADTs ●●●●●	case class, case object ●●●●○
empty ADTs ●●●●●	empty case classes ●●●●○
parameterized ADTs ●●●●●	parameterized ADTs ●●●●○
Proxy type: Data.Proxy ●●●●●	Custom proxy type ●●●●●
DataKinds and type-level ADTs ●●●●●	No DataKinds; custom type-level ADTs with empty parameterized ADTs ●●●○○
kinds ●●●●●	no native kinds; traits ●●●●○
GADTs ●●●●●	GADTs ●●●●○
multi-parameterized type classes ●●●●●	multi-parameterized traits ●●●●●

type families ●●●●○	match types ●●●○○
type-level lists ●●●●●	type-level tuples; custom type-level lists ●●●●○
type-level strings ●●●●○	type-level singleton strings ●●●●○

<Continue with the full version…>

Appendices

Appendix A

Typed Forms diagrams

I invented Typed Forms diagrams for a better illustration of type-level programming. They are not the only visual tool I invented for my books. In *Functional Design and Architecture*, I proposed a less formal yet useful set of diagrams for designing applications: necessity diagram, components diagram, and architecture diagram. These have some place in *pragmatic type-level design*, too, although I don't place an accent on this. Still, it was evident I needed something else, a visual language that could enrich the book and help the reader look at the code from a different angle. I wanted to create something that could compete with UML diagrams and be language-agnostic, enabling it for many programming ecosystems. Another consideration was that diagrams should be more formal, and there should be a possibility to generate code from fully formed diagrams automatically. I doubt we'll ever see a new interest in the CASE tools, which were arguably not successful enough, but having such a possibility would still be nice. We don't know how the industry will evolve over time, especially in the presence of AI, so I hope the format I developed here will at least provide you with aesthetic pleasure.

> **NOTE** I did my best to construct a diagram format that is language-agnostic, free from ambiguity, expressive

enough, and low-noise, but Typed Forms v.1.0 are not perfect, and there can be design flaws.

NOTE Usage of Typed Forms in the book may be occasionally inconsistent.

A.1 General conventions

The main model for the diagrams is Haskell's type system. If there are discrepancies between type systems from different languages, specific dialects of Typed Forms can be used.

Round-cornered shapes and sharp-cornered shapes

Round-cornered shapes. Represent type-related notions, including ordinary types (concrete types that don't have type parameters), type aliases, types of types (kinds), and others. Most of the shapes have rounded corners.

Example: `Int`, `String` (ordinary types); `*` (a "star" kind – a type of ordinary types).

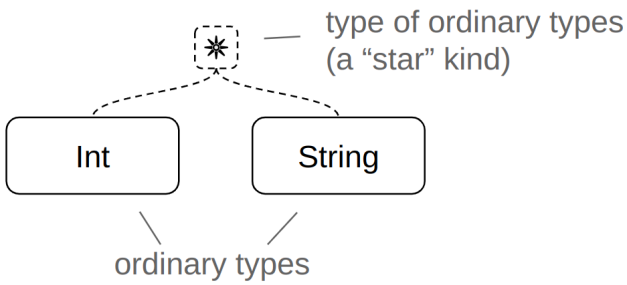

Figure A.1 Type-related notions

Sharp-cornered shapes. Represent values and value-related notions.

Example: `Animal` is an algebraic data type that is shaped with round corners, and `Cat` is a value constructor shaped with sharp corners. Also, `name` is a Cat's constructor field represented by a rectangle, a sharp-cornered shape.

```
data Animal = Cat
```

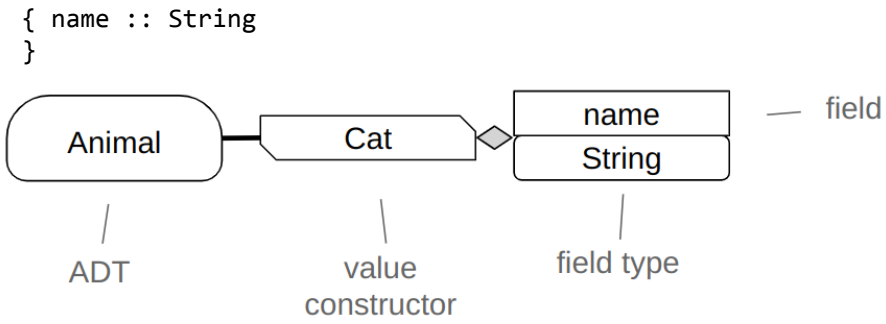

Figure A.2 Value-related notions

Solid line shapes and dashed line shapes

Solid line shapes are used for notions that can be directly encoded, such as types, value constructors, type parameters, and other similar concepts.

Dashed line shapes are reserved for kinds (types of types) and specific cases.

Arrows and connections

Connection lines without arrows mostly mean "has a relation to" or "notion has this type/kind."

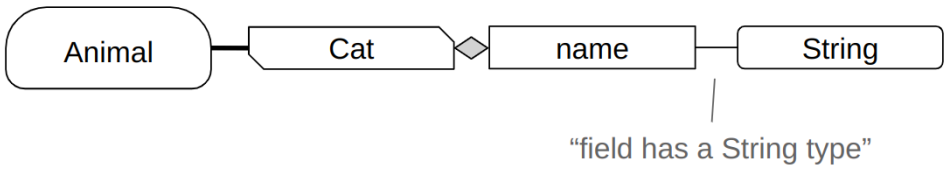

Figure A.3 Connection line for a type of field

Arrows from types to types mean "is part of." In general, arrows show a "flow" (or "delivery") of types, not dependency.

Example: the `String` type is included in the `Animal` ADT because `Animal` contains a field of the `String` type.

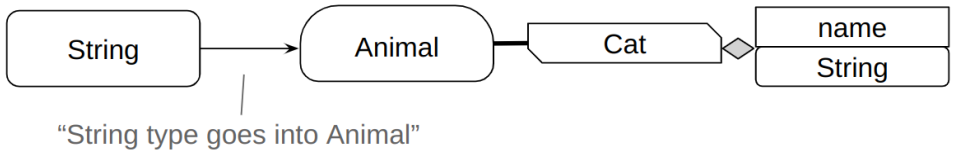

Figure A.4 "Delivery" of a type into another type

Optional elements

There is no obligation to put every possible aspect of a notion into a diagram. Many aspects are optional, for example, kinds of types. They can be omitted to preserve space.

However, sometimes important elements need to be hidden. To indicate this, a special notation is used; see figure A.5.

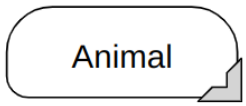

Figure A.5 Value constructors of the ADT are hidden.

Specific shapes and unknown type-related shapes

Many type-related notions have their own distinct shapes, including regular types, ADTs, type classes, and others. A regular round rectangle should represent unknown notions.

For example, a library exposes the type `Map`, but it's not known if this type is an ADT, a type alias, or something else. It should then be a rounded rectangle.

Comments

Comments on the diagrams should be represented by grey text and pointing lines.

A.2 Regular types, pairs, aliases, lists

Regular types

Regular types are represented by round rectangles of arbitrary sizes, but it's recommended to use 1-unit and 0.5-unit heights.

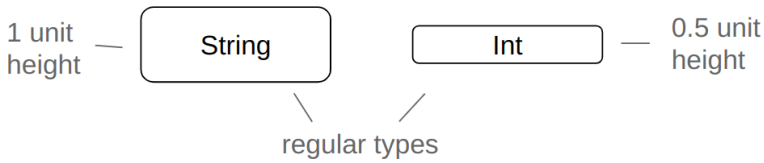

Figure A.6 Regular types

Pairs

Pairs are represented by a specific shape containing inner rectangles for field types separated by a rhombus delimiter shape.

```
(Int, String)
```

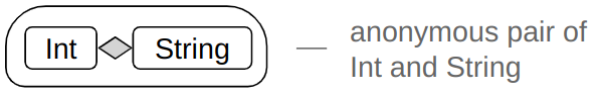

Figure A.7 Pair of Int and String

Aliases

Aliases can be represented by separate types that are pointed by a circle arrow.

A "burger" notation can also represent aliases.

```
type UserInfo = (Int, String)
```

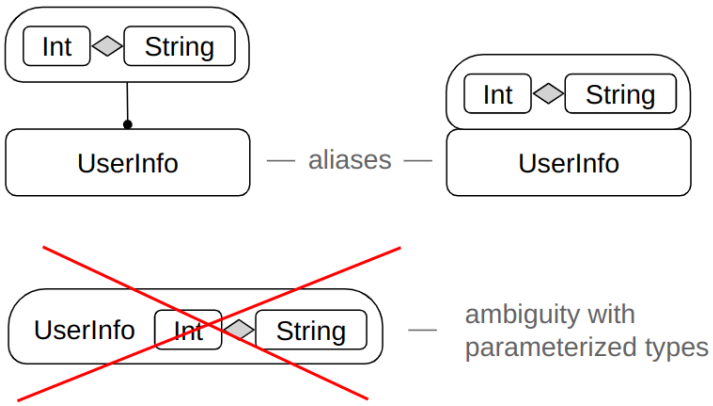

Figure A.8 Aliases

Lists

Lists have a special syntax: an extra rectangle over an enclosed type.

```
type UserCatalog1 = [(Int, String)]

type UserInfo = (Int, String)
type UserCatalog2 = [UserInfo]
```

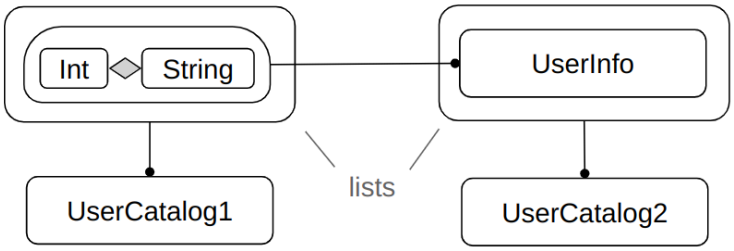

Figure A.9 Lists

A.3 Simple ADTs

Algebraic data types have a specific shape with rounded corners (similar to tuples, which are the unnamed half of ADT).

Value constructors have a specific shape with sharp corners.

Value constructors should be attached to the ADT with a thick solid line.

Fields should be specified as rectangles with types attached as separate rectangles or burgers.

Value constructors can be stacked or attached directly to the ADT.

```
data Animal
  = Cat { name :: String }
  | Dog { name :: String }
```

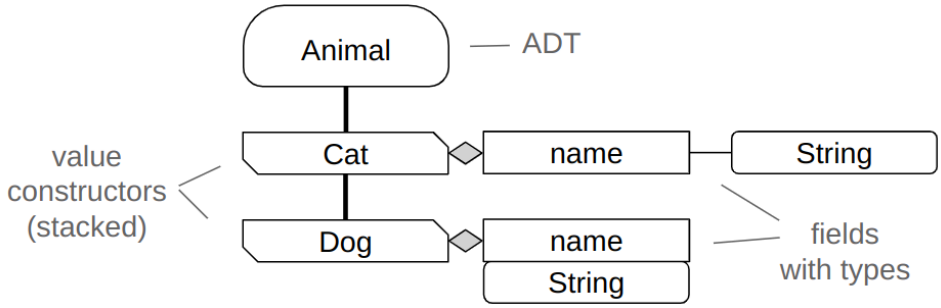

Figure A.10 ADT with stacked value constructors

```
data Hand = Left | Right
```

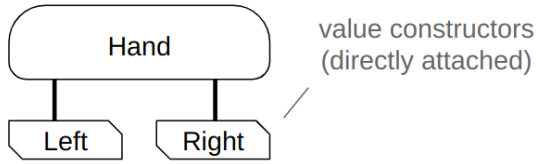

Figure A.11 ADT value constructors attached directly

Fields with no names can be denoted by a type using the rhombus delimiter.

```
type Id = Int
type Login = String
```

```
data User = User Id Login
```

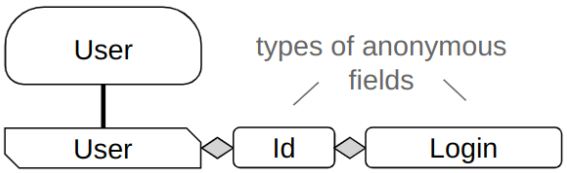

Figure A.12 Field having no name

It's possible to hide value constructors or fields. A special shape will indicate that the diagram is incomplete.

```
data Animal
  = Cat { name :: String }
  | Dog { name :: String }
```

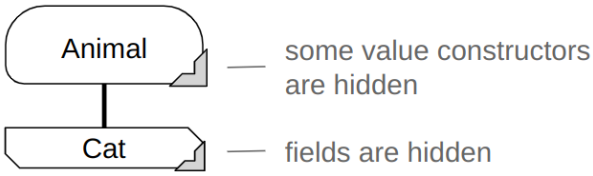

Figure A.13 Hidden value constructors and fields

Newtypes

Newtypes are an ADT-like wrapper type for any other type that has only one value constructor with one field. It is denoted by the ADT shape, which has a long-dashed line.

```
newtype LastName = LastName String
```

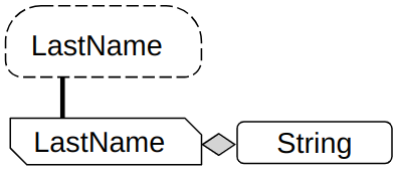

Figure A.14 Newtype

A.4 Parameterized types

Usual parameterized types

Parameterized types have one or more type variables.

Low-height nested rectangles represent type variables. The font size should be smaller than usual.

Generic (non-specific) type variables are formatted in italics and have the shape of an ellipse-like rectangle colored grey.

Specified type parameters have a usual rounded white rectangle shape, normal formatting, and carry the name of the type.

Examples in Haskell and C++:

```
Map Int v
```

```
template<int K, typename V>
class Map {};
```

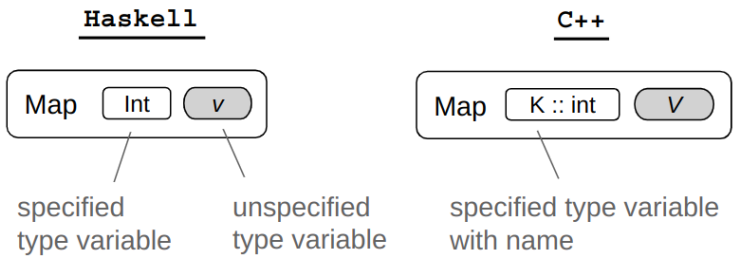

Figure A.15 Parameterized types

Simple syntax for parameterized aliases

Parameterized aliases can have independent variables and can borrow unspecified type variables. In simple syntax, all type variables from the source type should be either specified or borrowed.

```
-- Opaque type of unknown structure:
```

```
type Map k v1

type IntMap v2 = Map Int v2
```

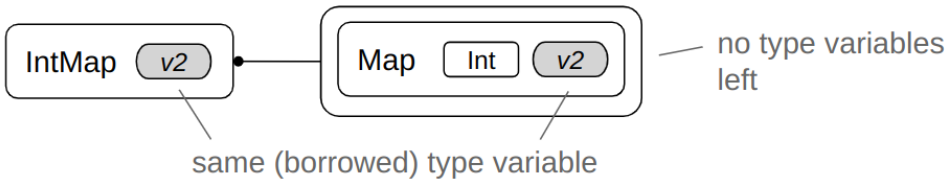

Figure A.16 Simple syntax for parameterized aliases

Complex parametrization with possible type variable renamings should be done with the generics specification syntax.

Abstract generic types

Abstract (opaque) generic types, if no additional information is known about them, may have a regular shape and italic font.

Figure A.17 Abstract generic type

A.5 Generics specification syntax

For complex cases of type variable specification, a notation of "delivery" can be used.

The circle-pointed arrow shows the direction of flow of a type variable. Note that the arrow starts on the parameterized type, not the type variable.

```
-- Opaque types of unknown structure:
type Parameterized v
type Specific

type Alias = Parameterized Specific
```

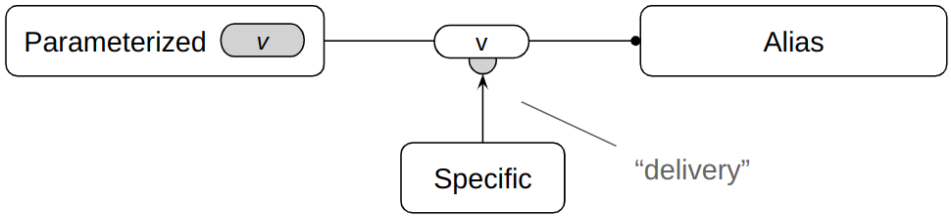

Figure A.18 Generics specification syntax

A.6 Type classes and instances

Type classes are represented by the ellipse rectangle with type variables syntax in grey. Type class names and type variable names should be italic.

Instances are denoted with a specific sharp-cornered rectangle. Specification of the types uses the generics specification syntax.

Type classes connect to instances using a dashed arrow. Note that the arrow starts on the type class itself, not on the type variable.

```
data LastName = LastName String

class Print a where
  print :: a -> IO ()

instance Print LastName where
  print (LastName n) = putStrLn n
```

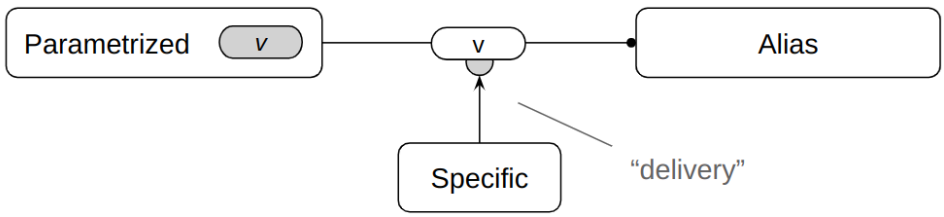

Figure A.19 Type classes, instances, and generics specification

A.7 Kinds

Kinds is a term in Haskell that denotes types of types.

> **NOTE** Kinds are obsolete in Haskell, but they are very representative, so they are used in the book.

All kinds should have dashed, round-cornered shapes.

Types connect to their kinds with a dashed line.

Ordinary types

Ordinary types have the "star" kind.

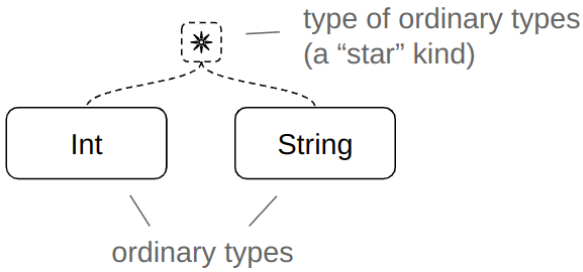

Figure A.20 "Star" kind

Specific kinds

Specific kinds have specific names.

```
data UserInfo (userId :: Nat) (login :: Symbol)
```

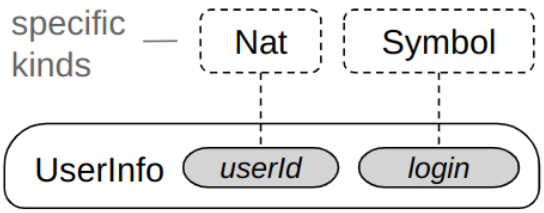

Figure A.21 Specific kinds

Type constructors

Type constructors (parameterized types) have complex kinds. Kinds can be connected with the grey rhombus shape to form a complex kind.

```
data Maybe a = Nothing | Just a
type MaybeAlias = Maybe
```

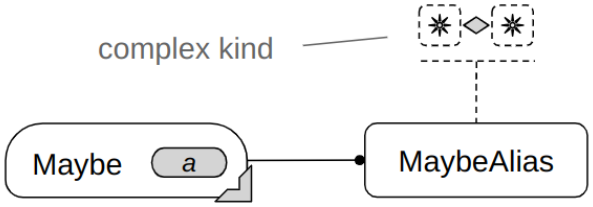

Figure A.22 Complex kind

Various kinds are presented in the following figure.

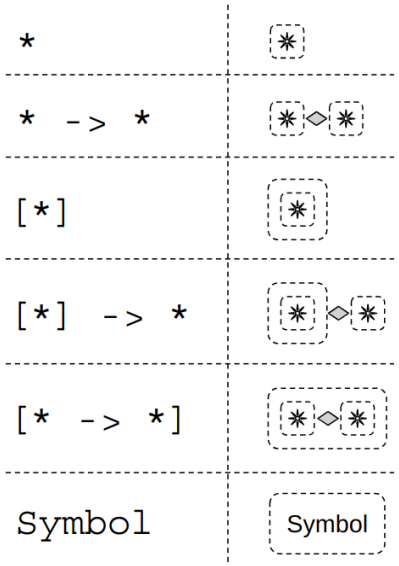

Figure A.23 Various kinds

A.8 Type promotion

In Haskell, types can be promoted one level up. Types become kinds, ADT value constructors become types of a specific kind.

```
{-# LANGUAGE DataKinds #-}

data PersonType = Person String String

data UserType = User
 { login :: Symbol
  , valid :: Bool
  , person :: PersonType
  }
```

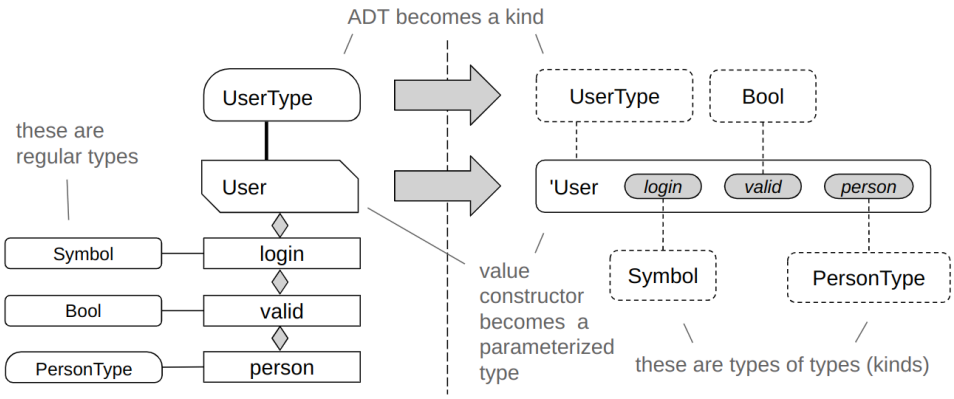

Figure A.24 Type promotion

A.9 Type families

A specially shaped round rectangle with a triangle denotes Haskell's type families. Open type families will have a white triangle, and closed type families will have a grey triangle.

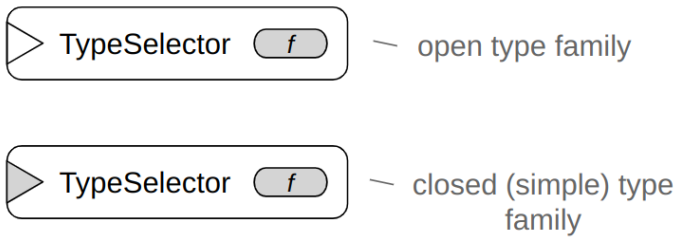

Figure A.25 Type families

A.10 HKD template

The HKD template from the Higher-Kinded Data design pattern is denoted with a specific "paper sheet" shape.

The specification of the HKD template may use the generic specification syntax.

```
{-# LANGUAGE TypeFamilies #-}

newtype GUID = GUID String

type family TypeSelector f where
  TypeSelector Int = Int
  TypeSelector GUID = GUID

data UserInfo f = User { userId :: TypeSelector f }
type MyUser = UserInfo Int
```

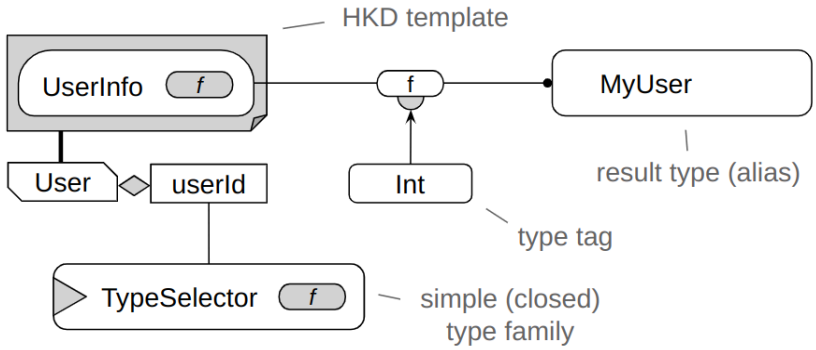

Figure A.26 The HKD design pattern