Type-Driven Development with Idris

Edwin Brady

MANNING

MEAP Edition
Manning Early Access Program

Type-Driven Development with Idris

Version 1

Welcome

Hello, and welcome to the MEAP for *Type Driven Development with Idris*. This is an exciting stage not only in the development of this book, but in the development of the Idris programming language itself, and I'm very pleased you've decided to buy this MEAP and join me in this project. Thanks!

This book is about making types work for you. Types are often seen as a tool for checking for errors, with the programmer writing a complete program first and using the type checker to detect errors. In type-driven development, we use types as a tool for constructing programs, using the type checker as our assistant, guiding us to a complete and working program.

I've been working with Idris and contributing to its development for several years now, and in this book I aim to explain the various type-driven programming techniques that I and others in the community have discovered and developed over the last few years.

To get the most out of this book, ideally you'll already be familiar with the basic ideas of functional programming, and have a good working knowledge of data structures such as lists and trees. If you understand concepts such as closures, recursive types and functions, higher order functions and programming with immutable data, then you should get on fine. No knowledge of any specific language is assumed.

At this stage, we're releasing the first two Chapters, which will get you up to speed with the main ideas behind type-driven development, and with the Idris language and environment. You'll learn about what it means to be a type, and how we treat a type as a plan for a program. You'll also learn about the built-in types and functions provided by Idris, and how to structure these into complete programs. If you're already familiar with a functional programming language, these chapters will get you up to speed on the basics of Idris before we dive more deeply into dependent types and type-driven development in the later chapters.

As you work through these early chapters, I'd love to hear from you! Your feedback is vital to making this book the best it can possibly be. Please use the Author Online forum to let me know if anything needs more explanation, if you think you've spotted a mistake, or even just to let me know how you're getting on.

Have fun!

—Edwin Brady

brief contents

1 Overview

This chapter covers

- Getting started with Idris
- An introduction to type-driven development
- The essence of pure functional programming

This book teaches a new approach to building robust software, *Type-driven Development*, using the Idris programming language. Traditionally, types are seen as a tool for checking for errors, with the programmer writing a complete program first and using either the compiler or run-time system to detect type errors. In type-driven development, we use types as a tool for constructing programs. We put the *type* first, treating it as a *plan* for a program, and use the compiler and type checker as our assistant, guiding us to a complete and working program which satisfies the type. The more expressive a type we give up front, the more confidence we have that the resulting program will be correct.

NOTE **Types and tests**
The name "Type-driven Development" suggests an analogy with Test-driven Development. There is a similarity, in that writing tests first helps establish a program's purpose and whether it satisfies some basic requirements. The difference is that unlike tests, which can usually only be used to show the *presence* of errors, types (used appropriately) can show their *absence*.

Even so, while types *reduce* the need for tests, they rarely eliminate it entirely.

Idris is a relatively young programming language, designed from the beginning to support type-driven development. A prototype implementation first appeared in 2008, with development of the current implementation beginning in 2011. It builds on decades of research into the theoretical and practical foundations of programming languages and type systems.

In Idris, types are a *first-class* language construct. Types can be manipulated, used, passed as arguments to functions and returned from functions just like any other value such as numbers, strings, or lists. This is a simple but powerful idea, which allows:

- relationships to be expressed between values, for example that two lists have the same length
- assumptions to be made explicit and checkable by the compiler, for example if we assume that a list is non-empty then Idris can ensure this assumption always holds *before the program is run*
- if desired, program behaviour to be formally stated and proven correct

In this chapter, I'll introduce the Idris programming language and give a brief tour of its features and environment. I'll also give an overview of type-driven development, discussing why types matter in programming languages, and how they can be used to guide software development. But first, it is important to understand exactly what we mean when we talk about "types".

1.1 What is a Type?

We are taught from an early age to recognise and distinguish *types* of objects. As a young child, you may have had a "shape sorter" toy. This consists of a box with various shaped holes in the top, as illustrated in Figure 1.1, and some shapes which fit through the holes. Sometimes, they are equipped with a small plastic hammer. The idea is to fit each shape (think of this as a "value") into the appropriate hole (think of this as the "type") possibly with coercion from the hammer.

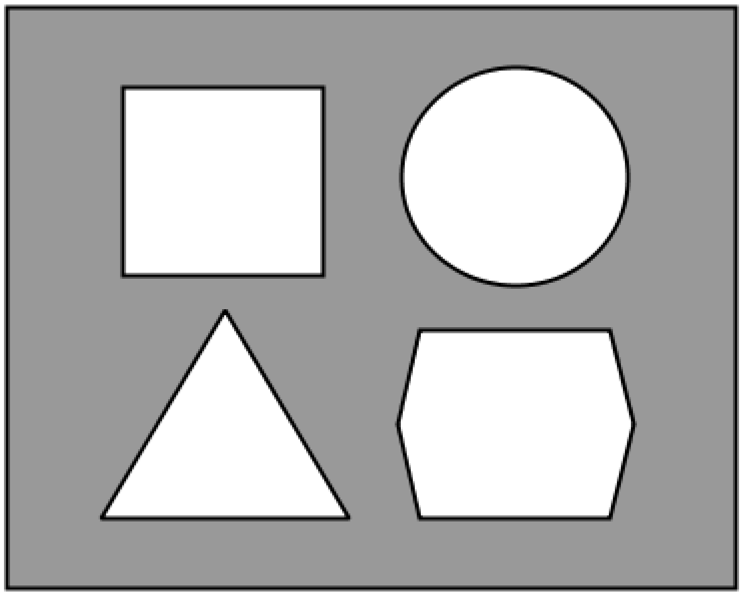


Figure 1.1 The top of a "shape sorter" toy. The shapes correspond the "types" of objects which will fit through the holes.

In programming, types are a means of classifying values. For example, the values `94`, `'a'` and `[1,2,3,4,5]` could respectively be classified as an integer, a character and a list of integers. All modern programming languages classify values by type, although they differ enormously in when and how they do so (e.g. whether they are checked statically at compile-time or dynamically at run-time, whether conversions between types are automatic or not, and so on.) Types serve several important roles:

1. For a *machine*, types describe how bit patterns in memory are to be interpreted.
2. For a *compiler or interpreter*, types help ensure that bit patterns are interpreted consistently when a program is executed.
3. For a *programmer*, types help name and organise concepts, aiding documentation and supporting interactive editing environments.

From our point of view in this book, the most important purpose of types is the third. Types help the programmer by allowing the naming and organisation of concepts (e.g. `square`, `circle`, `triangle` and `hexagon`), explicit documentation of the purpose of variables, functions and programs, and by driving code completion in an interactive editing environment. All modern statically typed languages support this to some extent, but as we will see, the expressivity of the Idris type system leads to powerful automatic code generation.

1.2 A quick tour of Idris

While this chapter is primarily concerned with introducing the concepts of type-driven development, we will be using the Idris programming language to explore these concepts, so we'll begin by taking a quick tour of the Idris system itself. It consists of an interactive environment and a batch mode compiler. In the interactive environment we can load and type check source files, evaluate expressions, search libraries, browse documentation and compile and run complete programs. We will explore these features in depth throughout this book.

In this section, I will briefly introduce the most important features of the environment, which are evaluation and type checking, and describe how to compile and run Idris programs. I will also introduce the two most distinctive features of the Idris language itself:

- *Holes*, which stand for incomplete programs.
- The use of types as a *first-class* language construct.

1.2.1 The interactive environment

Much of our interaction with Idris is through an interactive environment, called the "Read-Eval-Print Loop", typically abbreviated as the "REPL". As the name suggests, the REPL will *read* input from the user, usually in the form of an expression, *evaluate* the expression, then *print* the result. Once Idris is installed, you can start the REPL by typing `idris` at a shell prompt. You should see something like the following:

```
     ____    __     _
    /  _/___/ /____(_)____
    / // __  / ___/ / ___/     Version 1.0
  _/ // /_/ / /  / (__  )      http://www.idris-lang.org/
 /___/\__,_/_/  /_/____/       Type :? for help

Idris is free software with ABSOLUTELY NO WARRANTY.
For details type :warranty.
Idris>
```

NOTE **Installing Idris**
Full instructions on how to download and install Idris are given in Appendix A.

You can enter expressions to be evaluated at the `Idris>` prompt. For example, arithmetic expressions work in the conventional way, with the usual precedence rules (i.e. `*` and `/` have higher precedence than `+` and `-`):

```
Idris> 2 + 2
4 : Integer

Idris> 2.1 * 20
42.0 : Double

Idris> 6 + 8 * 11
94 : Integer
```

Or, we could manipulate `String`s. The `++` operator is used to concatenate `String` s, and the `reverse` function reverses a `String`:

```
Idris> "Hello" ++ " " ++ "World!"
"Hello World!" : String

Idris> reverse "abcdefg"
"gfedcba" : String
```

Notice that Idris prints not only the result of evaluating the expression, but also its *type*. In general, if you see something of the form `x : T`, i.e. some expression `x`, a colon, and some other expression `T`, this can be read as "`x` has type `T`."

So, above, for example:

- `4` has type `Integer`
- `42.0` has type `Double`
- `"Hello World!"` has type `String`

1.2.2 Checking Types

The REPL provides a number of *commands*, all prefixed by a colon. One of the most commonly useful is `:t`, which allows us to check the *types* of expressions, without evaluating them. For example:

```
Idris> :t 2 + 2
2 + 2 : Integer

Idris> :t "Hello!"
"Hello!" : String
```

Types, such as `Integer` and `String`, can be manipulated just like any other value. Therefore, we can check the types of `Integer` and `String` too:

```
Idris> :t Integer
```

```
Integer : Type

Idris> :t String
String : Type
```

It is natural to wonder what the type of `Type` itself might be. In practice, we never need to worry about this, but for the sake of completeness, let's take a look:

```
Idris> :t Type
Type : Type 1
```

That is, `Type` has type `Type 1`, then `Type 1` has type `Type 2`, and so on, as far as we are concerned, forever. The good news is that Idris will take care of such details for us.

1.2.3 Compiling and Running Idris Programs

In the end, as well as evaluating expressions and inspecting the types of functions, we would like to be able to compile and run complete programs. Listing 1.1 shows a minimal Idris program.

Listing 1.1 Hello, Idris World

```
module Main ❶

main : IO () ❷
main = putStrLn "Hello, Idris World!" ❸
```

❶ Module header
❷ Function declaration
❸ Function definition

At this stage, there is no need to worry too much about the syntax or how the program works. For now, it suffices to know that Idris source files consist of a module header and a collection of function and data type definitions. They may also import other source files. Here, the module is called `Main` and there is only one function definition, called `main`. The entry point to any Idris program is the function `main` in the module `Main`. To run the program:

1. Create a file called `hello.idr` in a text editor[1]. Idris source files all have the extension `.idr`

Footnote 1 I recommend Atom, because it has a mode for interactive editing of Idris programs which we will be using in this book, but any text editor is fine.

2. Enter the code in Listing 1.1
3. In the working directory where you saved `hello.idr`, start up an Idris REPL with the command `idris hello.idr`
4. At the Idris prompt, type `:exec`

If all is well, you should see something like the following:

```
$ idris hello.idr
     ____    __     _
    /  _/___/ /____(_)____
    / // __  / ___/ / ___/     Version 1.0
  _/ // /_/ / /  / (__  )      http://www.idris-lang.org/
 /___/\__,_/_/  /_/____/       Type :? for help

Idris is free software with ABSOLUTELY NO WARRANTY.
For details type :warranty.
Type checking ./hello.idr
*hello> :exec
Hello, Idris World
```

Here, `$` stands for your shell prompt. Alternatively, you can create a standalone executable by invoking the `idris` command with the `-o` option, as follows:

```
$ idris hello.idr -o hello
$ ./hello
Hello, Idris World
```

TIP **The REPL Prompt**

The REPL prompt, by default, tells you the name of the file currently loaded. The prompt `Idris>` indicates that no file is loaded, whereas the prompt `*hello>` indicates that the file `hello.idr` is loaded.

1.2.4 Incomplete Definitions: Working with Holes

Earlier, I compared working with types and values to inserting shapes into a shape sorter toy. Much as the square shape will only fit through a square hole, the argument `"Hello, Idris World!"` will only fit into a function in a place where a `String` type is expected.

Idris functions themselves can contain *holes*. A function with a hole is *incomplete*. Only a value of an appropriate type will fit into the hole, just as a square shape will only fit into a square hole in the shape sorter. Here is an incomplete implemention of the "Hello, Idris World!" program above:

```
module Main

main : IO ()
main = putStrLn ?greeting
```

If you edit `hello.idr` to replace the string `"Hello, Idris World!"` with `?greeting`, and load it into the Idris REPL, you should see something like the following:

```
Type checking ./hello.idr
Holes: Main.greeting
*hello>
```

Incomplete progams can't be compiled, but they can be *type checked*, and they can be evaluated at the REPL. The syntax `?greeting` introduces a *hole*, which is a part of the program still to be completed. When Idris encounters the hole `?greeting`, it creates a new name `greeting` which has a type but no definition. We can inspect the type using `:t` at the REPL:

```
*hello> :t greeting
--------------------------------------
greeting : String
```

If we try to evaluate it, on the other hand, Idris will show us that it is a hole:

```
*hello> greeting
?greeting : String
```

TIP **Reloading**

Instead of exiting the REPL and restarting, you can also reload `hello.idr` with the `:r` REPL command, as follows:

```
*hello> :r
Type checking ./hello.idr
Holes: Main.greeting
*hello>
```

Holes allow us to develop programs *incrementally*, writing the parts we know and asking the machine to help us by giving the types for the parts we don't. For example, let's say we would like to print a character (with type `Char`) instead of a `String`. The `putStrLn` function requires a `String` argument, so we can't simply pass a `Char` to it, as we have tried in Listing 1.2

Listing 1.2 A program with a type error

```
module Main

main : IO ()

main = putStrLn 'x'  ❶
```

❶ Type error, giving a character instead of a string.

If we try loading this program into the REPL, Idris will report an error:

```
hello.idr:4:17:When checking right hand side of main:
When checking an application of function Prelude.putStrLn:
        Type mismatch between
                Char (Type of 'x')
        and
                String (Expected type)
```

We have to convert a `Char` to a `String` somehow. Even if we don't know exactly how to achieve this at first, we can start by adding a hole to stand in place of a conversion.

```
module Main
```

```
main : IO ()
main = putStrLn (?convert 'x')
```

Then, we can check the type of the `convert` hole:

```
*hello> :t convert
--------------------------------------
convert : Char -> String
```

The type of the hole, `Char -> String` is the type of a function which takes a `Char` as an input and returns a `String` as an otutput. We'll discuss type conversions in more detail in Chapter 2, but an appropriate function to complete this definition is `cast`:

```
main : IO ()
main = putStrLn (cast 'x')
```

1.2.5 Types as a First-class Construct

A *first-class* language construct is one for which there are no *syntactic* restrictions on where or how it can be used. That is, it can be passed to functions, returned from functions, stored in variables, and so on.

In most statically typed languages, there are restrictions on where types can be used, and there is a strict syntactic separation between types and values. You can't, for example, say `x = int` in the body of a Java method. In Idris, there are no such restrictions, and types are first-class: not only can types be used in the same way as any other language construct, any construct can appear as part of a type. Listing 1.3 illustrates this with a small example.

Listing 1.3 Calculating a type, given a boolean value

```
stringOrInt : Bool -> Type
stringOrInt x = case x of
                     True => Int
                     False => String

getStringOrInt : (x : Bool) -> stringOrInt x  ❶
getStringOrInt x = case x of
                        True => 94  ❷
                        False => "Ninety four"  ❸

valToString : (x : Bool) -> stringOrInt x -> String  ❹
valToString x val = case x of
                         True => cast val  ❺
```

```
False => val ❻
```

❶ The return type is calculated from the value of the input
❷ The input x was True, so this needs to be an Int
❸ The input x was False, so this needs to be a String
❹ The argument type is calculated from the value of the input
❺ The input x was True, so the argument val must be an Int and needs to be converted to a String
❻ The input x was False, so the argument val must be a String and can be returned directly

NOTE **Function Syntax**
We will go into much more detail on Idris syntax in the coming chapters. For now it suffices to know:

- A *function* type takes the form `a -> b -> ... -> t`, where a, b, etc are the *input* types and `t` is the *output* type. Inputs may also be annotated with names.
- `name : type` declares a new function `name`, of type `type`.
- Functions are defined by *equations*, for example:

```
square x = x * x
```

This defines a function called `square` which multiplies its input by itself.

Here, `stringOrInt` is a function which computes a type. We then use this function in two ways:

- Firstly, in `getStringOrInt`, we use `stringOrInt` to calculate the return type. If the input is `True`, `getStringOrInt` returns an `Int`, otherwise it returns a `String`.
- Secondly, in `valToString`, we use `stringOrInt` to calculate an argument type. If the first input is `True`, the second input must be an `Int`, otherwise it must be a `String`.

We can see in some more detail what is going on by introducing holes in the definition of `valToString`:

```
valToString : (x : Bool) -> stringOrInt x -> String
valToString x val = case x of
                         True => ?xtrueType
                         False => ?xfalseType
```

Inspecting the type of a hole with `:t` gives us not only the type of the hole itself, but also the types of any local variables in scope. If we check the type of `xtrueType`, we'll see the type of `val` which is computed when `x` is known to be `True`:

```
*fctype> :t xtrueType
  x : Bool
  val : Int
--------------------------------------
xtrueType : String
```

So, if `x` is `True`, then `val` must be an `Int`, as computed by the `stringOrInt` function. Similarly, we can check the type of `xfalseType` to see the type of `val` when `x` is known to be `False`:

```
*fctype> :t xfalseType
  x : Bool
  val : String
--------------------------------------
xfalseType : String
```

This allows us to express relationships between function argument types and return types, in that the value of an argument can be used to compute the type of some other argument. If you've programmed in C, you'll have seen a similar idea with the `printf` function, where one argument is a "format string" which dictates the number and expected types of the remaining arguments. The C type system cannot check that the format string is consistent with the arguments, so this check is often hard-coded into C compilers. In Idris, however, a function similar to `printf` can be written directly by taking advantage of types as a first-class construct. There is a function to calculate a type from a format string, `printfTy`:

```
*printf> :t printfTy
printfTy : String -> Type

*printf> printfTy "The answer is: %d"
Int -> String : Type

*printf> printfTy "%s = %d"
String -> Int -> String : Type
```

Then the type of `printf` uses `printfTy` to compute the expected type from a given format string:

```
*printf> :t printf
printf : (f : String) -> printfTy f

*printf> printf "The answer is: %d" 42
"The answer is: 42" : String

*printf> printf "%s = %d" "Answer" 42
"Answer = 42" : String
```

We'll look at the complete definition of `printf` in Idris in Chapter 4. This idea, using a value of one argument to determine the type of another, comes up regularly when programming in Idris, and is common in software development in general. For example:

- A database schema determines the allowed forms of queries on a database.
- A form on a web page determines the number and type of inputs expected.
- A network protocol description determines the types of values which may be sent or received over a network.

In each of these cases, one piece of data tells us about the expected form of some other data. By encoding this relationship in the types, we can ensure that any necessary checks are made that the data is well formed, even if that data is received from user input or a network and therefore out of our control. We'll discuss this further in Chapter 9.

1.3 Introducing Type-driven Development

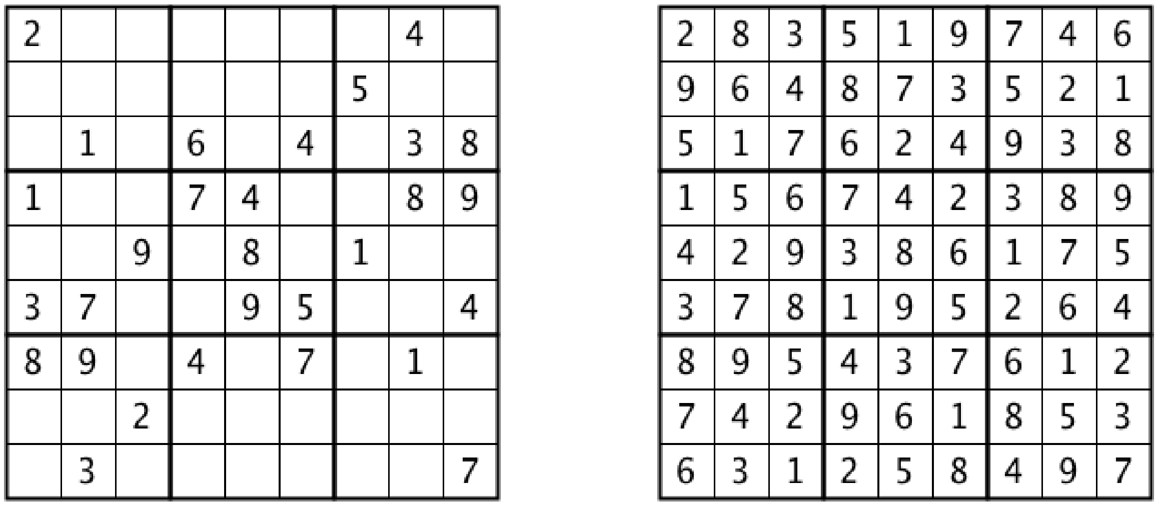
Type-driven Development is a style of programming in which a function's type is given first, and *incomplete* functions, possibly containing holes, may be type checked and gradually refined until they are complete. The overall process is to write a type, then fill in the details of a corresponding function until it is complete and satisfies the type, possibly refining and editing the type as necessary. Idris, with its support for holes and first-class types, strongly encourages this style of programming.

1.3.1 What is Type-driven Development?

Instead of thinking of types in terms of *checking*, with a programming language criticising us when we make a mistake, in type-driven development we think of types as being a *plan*, with the language's compiler acting as our guide leading us to a working, robust program. Starting with a type and an empty program, we gradually add details to the program until it is complete, regularly using the compiler to check that the program so far satisfies the type.

Type-driven development, in a sense, treats programming as solving a puzzle. The program itself is the solution to the puzzle, with the type describing the goal of the puzzle. When we solve puzzles, for example a sudoku or a jigsaw, we don't place all of the numbers or pieces at once then check that our result satisfies the rules. Rather, we begin with an empty grid with a few numbers as hints, or an empty table with a picture as a guide.

Figure 1.2 shows an example sudoku puzzle. The idea is to fill in the grid such that the digits 1 to 9 appear exactly once each in every row, column and 3*3 box. Several numbers are already filled in at the start. More difficult puzzles have fewer numbers filled in, but *all* sudoku puzzles have exactly one solution. On the left is the initial grid, and on the right the single valid solution.



2							4	
						5		
	1		6		4		3	8
1			7	4			8	9
		9		8		1		
3	7			9	5			4
8	9		4		7		1	
		2						
	3							7

2	8	3	5	1	9	7	4	6
9	6	4	8	7	3	5	2	1
5	1	7	6	2	4	9	3	8
1	5	6	7	4	2	3	8	9
4	2	9	3	8	6	1	7	5
3	7	8	1	9	5	2	6	4
8	9	5	4	3	7	6	1	2
7	4	2	9	6	1	8	5	3
6	3	1	2	5	8	4	9	7

Figure 1.2 A Sudoku puzzle and its solution. The initial grid, on the left, contains holes for the solution.

The rules of sudoku, then, are analogous to a function's *type*. A partly filled grid is analogous to an incomplete function, and a solved grid is analogous to a finished function. But, we can only take these analogies so far. Programming differs from puzzle solving in two important respects:

1. Firstly, a well-designed puzzle typically has only one solution, but in programming there is very often more than one. Type-driven development helps here, because the more precise the type, the more we limit the number of available solutions, and the more confident we can be that the program is a solution to the correct problem.
2. Secondly, with a puzzle we know the rules in advance, and we know what form a solution will take, so we can usually tell quickly whether a solution is

correct. In programming, and in type-driven development in particular, we need to define the rules ourselves. In the process of developing a solution, we may find we need to revise the rules.

If we think of programming as puzzle solving, with the final program being a solution to a puzzle with rules specified by the program's type, we begin to see the importance of types. If we see a 9*9 grid of numbers, we don't necessarily know that it's the solution to a sudoku puzzle without being told explicitly. Similarly, if we see a program without its type, we don't necessarily know what the program is intended to do. With type-driven development, therefore, it is at least as important to write clear and precise *types* as it is to write clear programs.

1.3.2 Types and Precision

When we talk of "precision" in types, we are talking in terms of the number of values which are classified into that type. A more precise type classifies fewer values. For example, the type "integers between 1 and 9" is more precise than the type "integers", because there is an additional constraint on the value the integer can take, so there are fewer values.

In the case of a sudoku puzzle, we can describe the solution in several ways with differing levels of precision. In order of increasing precision:

- A 9*9 grid of numbers
- A 9*9 grid of positive integers
- A 9*9 grid of positive integers between 1 and 9
- A 9*9 grid, with 9 3*3 boxes, containing positive integers between 1 and 9, where each integer appears exactly once in each row, column, and box

All of these are valid descriptions of a solution, in that all solutions satisfy all of the descriptions, but only the last is precise enough for a person who has never previously seen a sudoku puzzle to have a chance of solving one correctly. In fact, it's precise enough to capture all of the rules of Sudoku: *illegal states are not representable*. Idris allows us to write types which capture all of the above levels of precision.

Let's look at another example, from the game of Draughts (or Checkers.)

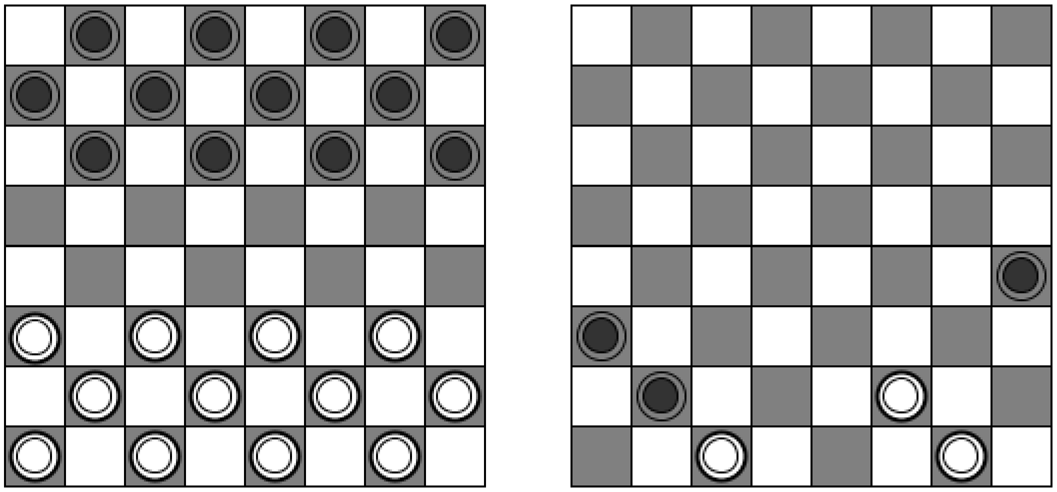


Figure 1.3 The starting position (left) and a later, valid position (right) in the game of Draughts, or Checkers

Figure 1.3 shows two positions. The position on the left is the starting position. Pieces may only move diagonally, thus the pieces will always be on the dark squares. Pieces are captured by "jumping" over opposing pieces, and the winner is the first player to capture all of their opponents pieces. Thus, the position on the right in Figure 1.3 is a valid position from later in a game. Again, when thinking about how to describe a position, we can use differing levels of precision. In increasing order of precision:

- A list of *(x, y)* coordinates paired with the colour of the piece at that coordinate. For example, taking the top left square as having the coordinate *(0, 0)* the position on the right in Figure 1.3 would be represented as the following list:

```
[((7, 4), Black), ((0, 5), Black), ((1, 6), Black),
 ((5, 6), White), ((2, 7), White), ((6, 7), White)]
```

- A list of *(x, y)* coordinates, in the range *0-7*, paired with the colour of the piece at that coordinate.
- A list of *(x, y)* coordinates, in the range *0-7*, paired with the colour of the piece at that coordinate, with no repetition of coordinates.
- Noticing that pieces can only be on the dark squares, and that the dark squares always have one even numbered coordinate and one odd numbered coordinate:

  A list of *(x, y)* in the range *0-7*, where one coordinate is even and the other odd, paired with the colour of the piece at that coordinate, with no repetition of coordinates.

In deciding how precise our types should be, we need to consider the following trade-off:

- More precise types increase our confidence that a program does what it is supposed to do.
- But if the type is *too* precise, the type itself becomes hard to understand and reduces our confidence that the type describes the right thing, as well as making maintenance and refactoring more difficult.

The most precise type for a sudoku solution would make it *impossible* to write a program which gave an incorrect solution, which would eliminate a large class of potential errors. Similarly, the most precise type for a Draughts position would make it *impossible* to write a program which played or accepted a move which led to an invalid position. On the other hand, these types would also make programs harder to write, understand, and maintain.

The challenge, then, is to find suitably descriptive types which help us arrive at working programs, without being so precise as to be hard to understand themselves! In type-driven development, we aim to choose a level of precision for a type which is appropriate to the task at hand and at a suitable level of abstraction for programmers to read and understand easily. Finding the right type is an iterative process, taking into account factors such as the importance of correctness, readability and maintainability. For example, we may err on the side of precision for an implementation of a secure communication protocol such as TLS (Transport Layer Security), but on the side of maintainability for a user interface.

Throughout this book, we will explore this trade-off. We will see how treating types as a first-class construct in Idris enables us to write precise and expressive types, and in particular how *dependent types* can help us write robust software.

1.3.3 Dependent Types

A type which is computed from (or parameterised by) some other value is called a *dependent type*. By computing a type from some other information, we can make types as precise as required. For example, some languages have a simple "List" type, describing lists of objects. This can be made more precise by parameterising over the element type: a generic "List of Strings" is more precise than simply a "List" and differs from a "List of Integers". Dependent types can be more precise still: a "List of 4 Strings" differs from a "List of 3 Strings".

Table 1.1 illustrates how types in Idris can have differing levels of precision even for fundamental operations such as appending lists. Given two specific input lists of Strings:

- `["a", "b", "c", "d"]`
- `["e", "f", "g"]`

We expect the following output list:

- `["a", "b", "c", "d", "e", "f", "g"]`

Using a *simple* type, both input lists have type `AnyList`, as does the output list. Using a *generic* type, we can specify that the input lists are both lists of strings, as is the output list. The more precise types mean that, for example, the output is clearly related to the input in that the element type is unchanged. Finally, using a *dependent* type, we can specify the sizes of the input and output lists. It is clear from the type that the length of the output list is the sum of the lengths of the input lists. That is, a list of 3 strings appended to a list of 4 strings results in a list of 7 strings.

Table 1.1 Appending specific typed lists. Unlike simple types, where there is no difference between the input and output list types, dependent types allow the length to be encoded in the type.

	Input ["a", "b", "c", "d"]	Input ["e", "f", "g"]	Output type
Simple	`AnyList`	`AnyList`	`AnyList`
Generic	`List String`	`List String`	`List String`
Dependent	`Vect 4 String`	`Vect 3 String`	`Vect 7 String`

NOTE **Lists and Vectors**

The syntax for the types in Table 1.1 is valid Idris syntax. Idris provides several ways of building list types, with varying levels of precision. In the table we can see two of these, `List` and `Vect`. `AnyList` is included in the table purely for illustrative purposes and is not defined in Idris.

`List` encodes generic lists with no explicit length, and `Vect` (short for "vector") encodes lists with the length explicitly in the type. We will see much more of both of these types throughout this book.

Table 1.2 illustrates how the input and output types of an `append` function can be written with increasing levels of precision in Idris. Using *simple* types, we write the input and output types as `AnyList`, suggesting that we have no interest in the types of the elements of the list. Using *generic* types, we write the input and output types as `List elem`. Here, `elem` is a *type variable* standing for the element types. Since the type variable is the same for both inputs and the output, the types specify that both input lists and the output list have a consistent element type. If we append two lists of integers, the types guarantee that the output will also be a list of integers. Finally, using *dependent* types, we write the inputs as `Vect n elem` and `Vect m elem` where `n` and `m` are variables representing the *length* of each list. The output type specifies that the resulting length will be the sum of the lengths of the inputs.

Table 1.2 Appending typed lists, in general. We use type variables to describe the relationship between the inputs and outputs, even though the exact inputs and outputs are unknown.

	Input 1 type	**Input 2 type**	**Output type**
Simple	`AnyList`	`AnyList`	`AnyList`
Generic	`List elem`	`List elem`	`List elem`
Dependent	`Vect n elem`	`Vect m elem`	`Vect (n + m) elem`

NOTE Types often contain *type variables*, like `n`, `m` and `elem` here. These are very much like parameters to generic types in Java or C#, but are so common in Idris that they have a very lightweight syntax. In general, concrete type names begin with an *upper case* letter, and type variable names begin with a *lower case* letter.

In the dependent type for the `append` function in Table 1.2, the parameters `n` and `m` are ordinary numeric values, and the `+` operator is the normal addition operator. All of these could appear in programs just as they have appeared here in the types. This is another example of first-class types in action: in a sense, the type of a specific `Vect` is computed from its length and element type.

1.3.4 Introductory Exercises

Throughout this book, I will give exercises which will help reinforce the concepts you have learned. As a warm up, I will give a selection of function specifications purely in the form of input and output types. For each of them, suggest possible operations that would satisfy the given input and output types. Note that there could be more than one answer in each case!

1. Input type: `Vect n elem`
   Output type: `Vect n elem`
2. Input type: `Vect n elem`
   Output type: `Vect (n * 2) elem`
3. Input type: `Vect (1 + n) elem`
   Output type: `Vect n elem`
4. Assume that `Bounded n` represents a number between zero and `n - 1`.
   Input types: `Bounded n`, `Vect n a`
   Output type: `a`

Suggested answers are available online.

1.4 Pure Functional Programming

Idris is a *pure functional* programming language. Before we begin exploring Idris in depth, therefore, we should understand what it means to be a *functional* language, and what we mean by the concept of *purity*. Unfortunately, there is no universally agreed definition of exactly what it means to be a *functional* programming language, but for our purposes we will take it to mean:

- Programs are composed of functions
- Program execution consists of evaluation of functions
- Functions are a first-class language construct

This differs from an *imperative* programming language primarily in that functional programming is concerned with the evaluation of expressions, rather than the execution of statements.

In a *pure* functional language, additionally:

- Functions may not have side effects such as modifying global variables, throwing exceptions or performing console input/output

- As a result, for any specific inputs, a function will *always* give the same result

You may wonder, very reasonably, how it is possible to write any useful software under these constraints. In fact, far from making it more difficult to write realistic programs, pure functional programming allows us to treat tricky concepts such as state and exceptions with the respect they deserve. Let's explore further!

1.4.1 Purity and Referential Transparency

The key property of a pure function is that the same inputs always produce the same result. This property is known as *referential transparency*. An expression (e.g. a function call) in a function is referentially transparent if it can be replaced with its result without changing the behaviour of the function. If functions produce only results, with no side effects, this propery is clearly true. Referential transparency is a very useful concept in type-driven development, because if a function has no side effects and is defined entirely by its inputs and outputs, then we can look at its input and output types and have a clear idea of the limits of what the function can do.

Figure 1.4 illustrates example inputs and outputs of the `append` function. It takes two inputs and produces a result, but there will be no interaction with a user such as reading from the keyboard, and no informative output such as logging or progress bars.

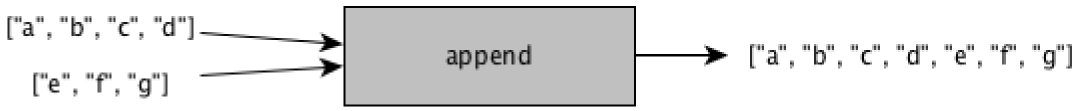


Figure 1.4 A pure function, taking inputs and producing outputs with no observable side effects

Figure 1.5 shows pure functions in general. There can be no observable side effects when running these programs, other than perhaps making the computer slightly warmer.

Inputs
Pure Function
Result

Figure 1.5 Pure functions, in general, take only inputs and have no observable side effects

Even so, pure functions are very useful in practice. It is possible to reason about their behaviour since the function always gives the same result for the same inputs, and they are important components for larger programs. Our `append` function is pure, and is a valuable component for any program which works with lists. It produces a list as a result, and because it is pure we know that it will not require any input, output any logging, or indeed delete any files.

1.4.2 Partial and Total Functions

Idris supports an even stronger property than purity for functions, making a distinction between *partial* and *total* functions. A *total* function is guaranteed to produce a result, meaning that it will return a value in a finite time for every possible well-typed input, and is guaranteed not to throw any exceptions. A *partial* function, on the other hand, might not return a result for some inputs. For example:

- The `append` function is total (assuming that lists cannot contain cycles) because it will always return a new list.
- The function which returns the first element of a list is *partial*, because it is not defined if the list is empty, and will therefore crash.

The distinction is important because knowing that a function is total allows us to make much stronger claims about its behaviour based on its type. If we have some function with a return type of `String`, for example, then we make different claims depending on whether the function is partial or total.

- If it is *total*: It will return a value of type `String` in finite time.
- If it is *partial*: *If* it does not crash, or enter an infinite loop, *then* the value it returns will be a `String`.

NOTE **Total and Partial Functions**
In most modern languages, we must assume that functions are partial and can therefore only make the latter, weaker, claim. Idris checks whether functions are total, and we can therefore often make the former, stronger, claim. For pragmatic reasons, above all that it is impossible to tell in general whether a function is total, it is one of the few distinctions we do not make in types. Nevertheless, as far as possible we will try to write *total* functions.

Realistically, programs must have side effects in order to be useful, and we are alway going to have to deal with unexpected or erroneous inputs in practical software. At first, this would seem to be impossible in a pure language. There is, however, a way: pure functions may not be able to perform side effects, but they can *describe* them.

1.4.3 Side Effecting Programs

Consider a function which reads two lists from a file, appends them, prints the resulting list and returns it. Listing 1.4 outlines this function in an imperative-style pseudocode, using simple types.

Listing 1.4 (Pseudocode) Appending lists read from a file

```
List appendFromFile(File h) {
    list1 = readListFrom(h)
    list2 = readListFrom(h)

    result = append(list1, list2)
    print(result)

    return result
}
```

This program takes a file handle as an input, and returns a `List`, with some side effects. It reads two lists from the given file and prints the list before returning. Figure 1.6 illustrates this, for the situation when the file contains the two lists `["a", "b", "c", "d"]` and `["e", "f", "g"]`.

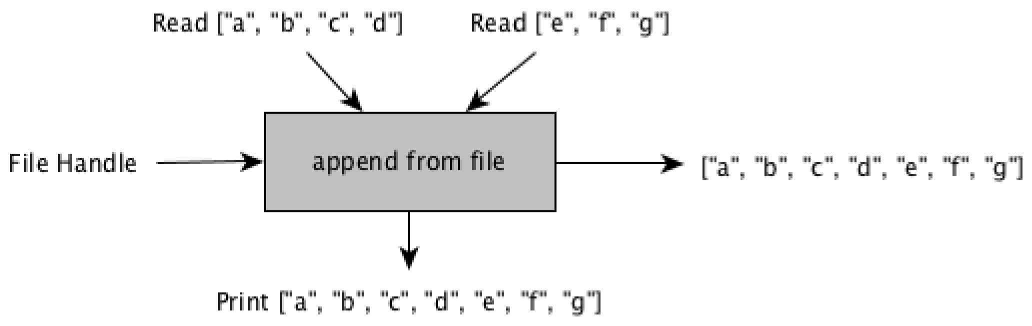

Read ["a", "b", "c", "d"]
Read ["e", "f", "g"]
File Handle
append from file
["a", "b", "c", "d", "e", "f", "g"]
Print ["a", "b", "c", "d", "e", "f", "g"]

Figure 1.6 A side effecting program, reading inputs from a file, printing the result and returning the result

The `appendFromFile` function does not satisfy the referential transparency property. Referential transparency requires that an expression can be replaced by its result without changing the program's behaviour. Here, however, replacing a call to `appendFromFile` with its result means that nothing will be read from the file, and nothing will be output to the screen. From the type-driven development perspective, the function's *input* and *output* types tell us that the input is a file and the output is a list, but nothing about the side effects the function may execute.

In pure functional programming in general, and Idris in particular, we solve this problem by writing functions which *describe* side effects, rather than functions which *execute* them, and deferring the details of execution to the compiler and run-time system. We will explore this in greater detail later; for now it is sufficient to recognise that a program with side effects has a type which makes this explicit, for example, there is a distinction between the following:

- `IO String` is the type of a program which results in a `String`, possibly performing some input and output as a side effect.
- `String` is the type of a program which results in a `String` and is guaranteed to perform *no* input or output as side effects.

In Idris, we take this idea much further, writing types which describe different side effects a program could have such as modifying an external state, throwing an exception or more specific system interactions such as communicating over a network or spawning a concurrent process. Indeed, a useful pattern in type-driven development is to write a type which precisely describes the valid states of a system and constrains the operations the system is allowed to perform. Let's look at an example which illustrates this idea, and type-driven development with dependent types more generally: a simple vending machine.

1.5 Dependent Types In Action: Modelling a Vending Machine

In type-driven development, we try to write types which describe the important properties of a system and the assumptions which are made in the system. This serves several purposes:

- It provides clear documentation of the intention of individual functions in the system.
- It forces any assumptions made in the system to be *explicit* and *checkable*. If any of these assumptions are violated, the compiler will report a type error.
- Precise types help guide the programmer to a correct implementation.

In this section, I will describe the types involved in the representation of a simple vending machine system, and show how this can benefit from the type-driven approach. At this stage, we are concerned only with *types,* and not *implementation*.

1.5.1 Vending machine description

To keep things simple, our vending machine only accepts one type of coin (let's say, a pound coin) and dispenses one product (let's say, a chocolate bar). The machine contains *credit*. Inserting a pound coin increases the credit in the machine, and dispensing a chocolate bar reduces the credit.

We have several requirements, from the point of view of both the machine owner and the customer, informally stated as follows:

1. The machine must never dispense chocolate if there is no credit.
2. The machine must dispense a chocolate bar on request when there is credit available.
3. The machine must return coins on request when there is credit available.
4. The machine should only accept coins if there is chocolate available.

The first requirement is important to the machine owner, since they do not want to be giving away chocolate for free. The second and third requirements are important to the customer, because they do not want to lose coins, and to the owner, because taking coins from the customer without dispensing chocolate is bad for their reputation and increases their customer service workload. The final requirement improves usability, to avoid the frustration of inserting coins and finding the machine is empty.

Table 1.3 shows the basic operations of the machine, detailing the *state* of the machine before and after each operation. The state consists of the number of coins the machine has in credit, and the number of chocolate bars available. The state *before* can be thought of as a *precondition* on the operation, meaning that the operation is only valid if this precondition is satisfied. The state *after* can be thought of as a *postcondition* on the operation, meaning that the state is guaranteed to take this form after the operation is executed.

In the table, *pounds* and *chocs* are variables which are part of the state. They are non-negative integers so, for example, if the "Coins" state reads *1 + pounds,* this means there must be at least one pound coin in the machine.

Table 1.3 Vending Machine Operations, with input and output states

Coins (Before)	**Chocolate (Before)**	**Operation**	**Coins (After)**	**Chocolate (After)**
pounds	*1 + chocs*	Insert Coin	*1 + pounds*	*1 + chocs*
1 + pounds	*1 + chocs*	Vend Chocolate	*pounds*	*chocs*
pounds	*chocs*	Return Coins	*0*	*chocs*

Additionally, our machine supports operations for displaying messages to the user, for example displaying the credit available or errors such as "Sold out". Finally, it should be possible to maintain the machine, refilling it with more chocolate.

1.5.2 A Type-driven Vending Machine

In Idris, one way to model the vending machine would be with a type which explicitly captures the important parts of the machine state, the coins in credit and chocolates available. Then, we write *pure* functions for representing each operation to transition between machine states. For example, we might write the following type:

```
Machine 2 94
```

This refers to a vending machine which contains *2* coins, and has *94* chocolates available. In general, we write `Machine pounds chocs` as the type of a vending machine with `pounds` coins of credit and `chocs` chocolates available.

Table 1.4 Vending Machine Operations, with Idris types

Operation	**Input Types**	**Output Type**
Initialise	None	`Machine 0 0`
Insert Coin	`Machine pounds (1 + chocs)`	`Machine (1 + pounds) (1 + chocs)`
Vend Chocolate	`Machine (1 + pounds) (1 + chocs)`	`Machine pounds chocs`
Return Coins	`Machine pounds chocs`	`Machine 0 chocs`
Display Message	`String`, `Machine pounds chocs`	`IO (Machine pounds chocs)`
Refill Machine	`Nat` (named `bars`), `Machine 0 chocs`	`Machine 0 (bars + chocs)`

Table 1.4 describes the input and output types of the operations. These types can be written directly in an Idris program, as shown in Listing 1.5.

Listing 1.5 Vending Machine Interface, as Idris types

```
init       : Machine 0 0   ❶
insertCoin : Machine pounds (1 + chocs) ->
```

```
                   Machine (1 + pounds) (1 + chocs) ❷
vend             : Machine (1 + pounds) (1 + chocs) ->
                   Machine pounds chocs ❸
getChange        : Machine pounds chocs ->
                   Machine 0 chocs ❹
display          : String -> Machine pounds chocs ->
                   IO (Machine pounds chocs) ❺
refill           : (bars : Nat) -> Machine 0 chocs ->
                   Machine 0 (bars + chocs) ❻
```

❶ init returns a new vending machine state, initialised with 0 pounds and 0 chocolates.
❷ insertCoin takes a machine state as input, and returns a new machine state with one more pound coin.
❸ vend takes a machine state with at least one pound and at least one chocolate, and returns a new machine state with one fewer pound coin and one fewer chocolate.
❹ getChange takes any machine state and returns a machine state with no coins.
❺ display takes any machine state, and returns the same machine state, possibly executing some input or output as side effects.
❻ refill takes a number of chocolate bars and a machine with no coins, and returns a new machine state with chocolate bars added.

At this stage, you should at least be able to see how the initial specification leads to the types of the operations, and how those types clarify questions about the interface. Specific things to note about these types are:

- `Nat`, in Idris, is the type of *non-negative* numbers. It is an abbreviation of "natural number"[2] and is often used to represent the size of a data structure.

Footnote 2 Mathematicians may argue over whether zero is a natural number or not. In Idris, zero *is* a natural number!

- Each of the operations takes a machine specification as an input, and gives an updated machine specification as an output. Additionally, displaying a message may perform some I/O.
- The rules we specified in Table 1.3 have been written explicitly in the types. For example, it is explicit in the type that vending a chocolate bar will reduce both the credit and the number of chocolate bars available.
- When refilling the machine, it is clear from the machine state in the output type that the input `bars` in the number of bars to be added, rather than the total number of bars required.
- Most of the operations on the state are *pure* and this is reflected in the output types. The exception is `display`, which will output a `String`, so its output type

reflects this with `IO`.

This is not the whole story, however. We still have to ensure that the operations are *composed* correctly. That is, the output of one operation is correctly passed to the input of the next. There are several ways to achieve this, and we will revisit this example in Chapter 9.

1.5.3 Type, Define, Refine: The Process of Type-driven Development

The type-driven approach to the vending machine interface involves writing types which give a suitably precise *abstraction* of the machine's state which captures the requirements discussed informally in section 1.5.1.

Having decided that the type for the representation of `Machine` should include the number of coins and chocolate bars in the machine, the input and output types of the operations became informative enough to capture these requirements. The process of type-driven development is, therefore, as much about deciding how expressive types should be as writing functions which satisfy these types. Typically, we arrive at appropriate types via an iterative process. For the vending machine, this might proceed as follows:

1. Identify that we need to represent the vending machine.
2. Design a type `Machine`, initially with no parameters.
3. Choose the `vend` operation to implement.
4. Write down the input type `Machine` and the output type, also `Machine`. At this stage, we decide that they are precise enough.
5. Implement the function.
6. Notice that the function is not total: the input type `Machine` allows states where there is no money, no chocolate, or neither, and the output type `Machine` does not capture the possibility of error.
7. Refine the type to `Machine pounds chocs`, as in our example, recheck the `vend` function, notice that it is now total and move on to the next operation.

There are other reasonable refinements we could make, for example we might parameterise `Machine` by boolean flags indicating whether the machine was empty of coins or chocolate, which may be entirely adequate depending on the level of precision desired in the type.

In general, the process is:

1. Identify the data which is to be represented.
2. Design a type to represent that data, deciding which aspects of the data should

be part of the type.

3. Identify an operation to implement on that data.
4. Write down its input and return types, and decide whether they encode enough precision. If not, return to step 2.
5. Implement a function which satisfies the type. If this proves difficult, reconsider whether the type is appropriate, and return to step 4. For example, are you trying to use side effects which the type does not allow?
6. Check whether the implementation is a *total* function. If not, can the type be refined in order that it is total? For example, should the return type encode the possibility of error (return to step 4), or should the input type be more restricted (return to step 2)?
7. Identify the next operation, returning to step 3.

To put it even more succinctly, we can summarise type-driven development as an iterative process of "Type, Define, Refine":

1. Write a *Type* to represent the system we are modelling
2. *Define* a function over that type
3. *Refine* the type as necessary to capture any missing properties, or to help ensure totality

1.6 Summary

- Types are a means of classifying values. Programming languages use types to decide how to lay out data in memory, and to ensure that data is interpreted consistently.
- A type can be viewed as a *specification*, so that a language implementation (specifically, a type checker) can check whether a program conforms to that specification.
- In type-driven development, a type is viewed more like a *plan*, helping an interactive environment guide a programmer to a working program.
- Dependent types allow us to give more precise types to programs, and hence more informative plans for the machine.
- Idris is a programming language which is specifically designed to support type-driven development. It is a purely functional programming language, with first class dependent types.
- Idris allows programs to contain *holes* which stand for incomplete programs.
- In a functional programming language, program execution consists of evaluation of functions.
- In a *purely* functional programming language, additionally, functions have no *side effects*.
- Instead of writing programs which *perform* side effects, we write programs which *describe* side effects, with the side effects stated explicitly in a program's type.
- A *total* function is guaranteed to produce a result for any input in finite time.
- Type-driven development is an iterative process of "Type-Define-Refine", creating a type to model a system then defining functions and refining the types as necessary.