Programming in System Fω using Dhall

Programming in System Fω using Dhall

by Sergei Winitzki, Ph.D.

Version of November 29, 2025

Published by lulu.com in 2025

Contents

0 Preface

This book is an advanced-level tutorial on Dhall[1] for software engineers already familiar with the functional programming (FP) paradigm, as practiced in languages such as OCaml, Haskell, Scala, and others.

Dhall is positioned as an open-source language for programmable configuration files. Its primary design goal is to replace various templating languages for JSON, YAML, and other configuration formats, providing a powerful and safe programming language. The "Design choices" document[2] discusses some other considerations behind the design of Dhall.

This book views Dhall as:

- a powerful templating system for flexible and strictly validated configuration files in JSON, YAML, and other text-based formats;
- a formally specified and tested interpreter for a small purely functional programming language, useful for studying various language-independent aspects of advanced functional programming.

The book focuses on the last use case.

Although most of the code examples are in Dhall, much of the material of the book has a wider applicability. The book studies a certain flavor of purely functional programming without side effects and with guaranteed termination, which is known in the academic literature as "System Fω". That type system is the foundation of Haskell, Scala, and other advanced functional programming languages.

From the point of view of programming language theory, Dhall implements System Fω with some additional features, using a Haskell-like syntax.

For a more theoretical introduction to various forms of typed lambda calculus, System F, and System Fω, see:

- D. Rémy. Functional programming and type systems.[3]
- Lectures on Advanced Functional Programming, Cambridge, 2014-2015[4], in particular the notes on lambda calculus.[5]

[1] https://dhall-lang.org
[2] https://docs.dhall-lang.org/discussions/Design-choices.html
[3] https://gallium.inria.fr/~remy/mpri/
[4] https://www.cl.cam.ac.uk/teaching/1415/L28/materials.html
[5] https://www.cl.cam.ac.uk/teaching/1415/L28/lambda.pdf

Most of that theory is beyond the scope of this book, which is focused on issues arising in practical programming. The book contains many code examples, which have been validated automatically by the Dhall interpreter.

The Appendix of the book contains some theoretical material that proves the correctness of certain code constructions, notably the Church encodings of fixpoint types and the parametricity properties of existential types.

The text of this book was written and edited without using any LLMs.

1 Overview of Dhall

This book corresponds to the Dhall standard 23.0.0[1] and Dhall version `1.42.2`.

Dhall is a small, purely functional language. It will be easy to learn Dhall for readers already familiar with functional programming.

The syntax of Dhall is similar to that of Haskell. One major difference is Dhall's syntax for functions, which resembles the notation of System F and System Fω. Namely, System F's notation $\Lambda t.\ \lambda(x : t).\ f\ t\ x$ and System Fω's notation $\lambda(t : *).\ \lambda(x : t).\ f\ t\ x$ correspond to the Dhall syntax `λ(t : Type) → λ(x : t) → f t x`.

Here is an example of a complete Dhall program:

```
let f = λ(x : Natural) → λ(y : Natural) → x + y + 2
let id = λ(A : Type) → λ(x : A) → x
in f 10 (id Natural 20)
  -- This is a complete program; it evaluates to 32 of type Natural.
```

See the Dhall cheat sheet[2] for more examples of basic Dhall usage.

The Dhall standard prelude[3] defines a number of general-purpose functions such as `Natural/lessThan` or `List/map`.

1.1 Identifiers and variables

Dhall variables are immutable constant values with names, introduced via the "`let`" syntax. We will call them "variables", even though they stand for constants that cannot vary.

For example, `let x = 1 in ...` defines the variable `x` that can be used in the code that follows. Names of variables are arbitrary identifiers, like in most programming languages.

Identifiers in Dhall may contain dash and slash characters. Examples of valid identifiers are `List/map` and `start-here`.

The slash character is often used in Dhall's standard library, providing suggestive function names such as `List/map`, `Optional/map`, etc. However, Dhall does not treat those names specially. It is not required that functions working with `List` should have names such as `List/map` or `List/length`.

Identifiers with dashes can be used, for example, as record field names, as it is sometimes needed in configuration files:

[1] https://github.com/dhall-lang/dhall-lang/releases/tag/v23.0.0
[2] https://docs.dhall-lang.org/howtos/Cheatsheet.html
[3] https://prelude.dhall-lang.org/

```
⊢ { first-name = "John", last-name = "Reynolds" }

{ first-name = "John", last-name = "Reynolds" }
```

However, identifiers may not *start* with a dash or a slash character.

Identifiers may contain arbitrary characters (even keywords or whitespace) if escaped in backquotes.

```
⊢ let `: a b c` = 1 in 2 + `: a b c`

3

⊢ let `forall` = 3 in `forall`

3
```

The standalone underscore character (_) is used in Haskell, Scala, other languages as syntax for a special "unused" variable. But in Dhall, the symbol _ is a variable like any other:

```
⊢ let _ = 123 in _ + _

246
```

Of course, one might still use the symbol _ in Dhall code as a convention for unused variables. However, the Dhall interpreter will not treat the variable _ in any special way and will not verify that the variable _ actually remains unused.

1.2 Primitive types

Integers must have a sign: for example, `+1` or `-1` have type `Integer`. The integer values `-0` and `+0` are the same.

`Natural` numbers must have no sign (for example, `123`).

Values of types `Natural` and `Integer` have unbounded size. There is no overflow. Dhall does not have built-in 32-bit or 64-bit integers with overflow, as it is common in other programming languages.

Dhall supports other numerical types, such as `Double` or `Time`, but there is little one can do with values of those types (other than print them). For instance, Dhall does not directly support floating-point arithmetic on `Double` values, or arithmetic operations with dates and times.

Strings have type `Text` and support interpolation:

```
⊢ let x = 123 in "The answer is ${Natural/show x}."

"The answer is 123."
```

Strings also support search/replace and concatenation, but no other operations.

For instance, it is currently not possible in Dhall to compare strings lexicographically, or even to compute the length of a string.

These restrictions are intentional and are designed to keep users from writing Dhall programs whose behavior is based on analyzing string values, such as parsing external input or creating "stringly-typed" data structures.

Nevertheless, it is possible in Dhall to "assert" certain conditions that string values should satisfy. For example, one can assert that a string value should be non-empty, or begin with certain characters, or consist entirely of alphanumeric characters. These and other similar conditions may be enforced statically via the "assert" feature that we will discuss later. A Dhall program will fail to compile if an asserted condition does not hold.

1.3 Functions and function types

An example of a function in Dhall is:

```
let inc = λ(x : Natural) → x + 1
```

Dhall does not support Haskell's concise function definition syntax such as `inc x = x + 1`, where arguments are given on the left-hand side and types are inferred automatically. Functions must be defined via `λ` symbols as we have just seen.

Generally, functions in Dhall look like `λ(x : input_type) → body`, where:

- `x` is a bound variable representing the function's input argument,
- `input_type` is the type of the function's input argument, and
- `body` is an expression that may use `x`; this expression computes the output value of the function.

A function's type has the form `∀(x : input_type) → output_type`, where:

- `input_type` is the type of the function's input value, and
- `output_type` is the type of the function's output value.

The function `inc` shown above has type `∀(x : Natural) → Natural`.

The Unicode symbols `∀`, `λ`, and `→` may be used interchangeably with their equivalent ASCII representations: `forall`, `\`, and `->`. In this book, we will use the Unicode symbols for brevity.

Usually, the function body is an expression that uses the bound variable `x`. However, the type expression `output_type` might itself also depend on the input *value* `x`. We will discuss such functions in more detail later. In many cases, the output type does not depend on `x`. Then the function's type may be written in a simpler form: `input_type → output_type`.

For example, the function `inc` shown above may be written with a type annotation like this:

```
let inc : Natural → Natural = λ(x : Natural) → x + 1
```

We may also write a fully detailed type annotation if we like:

```
let inc : ∀(x : Natural) → Natural = λ(x : Natural) → x + 1
```

It is not required that the name `x` in the type annotation (`∀(x : Natural)`) should be the same as the name `x` in the function (`λ(x : Natural)`). So, the following code is just as valid (although may be confusing):

```
let inc : ∀(a : Natural) → Natural = λ(b : Natural) → b + 1
```

1.3.1 Curried functions

All Dhall functions have just one argument. To implement functions with more than one argument, one can use either curried functions or record types (see below).

For example, a function that adds 3 numbers can be written in different ways according to convenience:

```
let add3_curried : Natural → Natural → Natural → Natural
  = λ(x : Natural) → λ(y : Natural) → λ(z : Natural) → x + y + z

let add3_record : { x : Natural, y : Natural, z : Natural } → Natural
  = λ(record : { x : Natural, y : Natural, z : Natural }) →
    record.x + record.y + record.z
```

Most functions in the Dhall standard library are curried. Currying allows function argument types to depend on some of the previous curried arguments.

1.3.2 Functions at type level

Types in Dhall are treated somewhat similar to values of type `Type`. For example, the symbol `Natural` is considered to have type `Type`:

```
⊢ Natural

Natural : Type
```

Because of this, we can define functions whose input is a type (rather than a value). The output of a function can also be a type, or again a function having types as inputs and/or outputs, and so on.

We will discuss functions of types in more detail later in this chapter. For now, we note that an argument's type may depend on a previous curried argument, which can be itself a type. This allows Dhall to implement a rich set of type-level features:

- Functions with type parameters: for example, `λ(A : Type) → λ(x : A) → ...`
- Type constructors, via functions of type `Type → Type` (both the input and the output is a type).
- Type constructor parameters: for example, `λ(F : Type → Type) → λ(A : Type) → λ(x : F A) → ...`
- Dependent types: functions whose inputs are values and outputs are types.

1.3.3 Shadowing and the syntax for de Bruijn indices

De Bruijn indices are numbers that disambiguate shadowed variables in function bodies.

Consider a curried function that shadows a variable:

```
let _ = λ(x : Natural) → λ(x : Natural) → 123 + x
```

The inner function shadows the outer `x`; inside the inner function body, `123 + x` refers to the inner `x`. The outer `x` is a **shadowed variable**.

In most programming languages, one cannot directly refer to shadowed variables. Instead, one needs to rename the inner `x` to `t`:

```
let _ = λ(x : Natural) → λ(t : Natural) → 123 + t
```

Now the function body could (if necessary) refer to the outer `x`.

What if we do not want (or cannot) rename `x` to `t` but still want to refer to the outer `x`? In Dhall, we write `x@1` for that:

```
let _ = λ(x : Natural) → λ(x : Natural) → 123 + x@1
```

The `1` in `x@1` is called the **de Bruijn index** of the outer variable `x`.

De Bruijn indices are non-negative integers. We usually do not write `x@0`, we write just `x` instead.

Each de Bruijn index points to a specific nested lambda in some outer scope for a variable with a given name. For example, look at this code:

```
let _ = λ(x : Natural) → λ(t : Natural) → λ(x : Natural) → 123 + x@1
```

The variable `x@1` still points to the outer `x`. The presence of another lambda with the argument `t` does not matter for counting the nesting depth for `x`.

It is invalid to use a de Bruijn index that is greater than the total number of nested lambdas. For example, these expressions are invalid at the top level (as there cannot be any outer scope):

```
λ(x : Natural) → 123 + x@1  -- Where is x@1 ???

λ(x : Natural) → λ(x : Natural) → 123 + x@2  -- Where is x@2 ???
```

At the top level, these expressions are just as invalid as the expression `123 + x` because `x` remains undefined.

The variable `x` in the expression `123 + x` is considered a "free variable", meaning that it should have been defined in the outer scope.

Similarly, `x@1` in `λ(x : Natural) → 123 + x@1` is a free variable.

At the top level, all variables must be bound. One cannot evaluate expressions with free variables at the top level. Expressions with free variables must be within bodies of some functions that bind their free variables.

In principle, nonzero de Bruijn indices could be always eliminated by renaming some bound variables. The advantage of using de Bruijn indices is that we can refer to every variable in every scope without need to rename any variables.

To illustrate the usage of de Bruijn indices, consider how we would proceed with a symbolic evaluation of a function applied to an argument.

When a function of the form `λ(x...) → ...` is applied to an argument, we will need to substitute that `x`. It turns out that we may need to change some de Bruijn indices in the function body.

As an example, suppose we would like to evaluate a function applied to an argument in this code:

```
( λ(x : Natural) → λ(y : Natural) → λ(x : Natural) → x + x@1 + x@2 ) y
```

This expression has free variables `y` and `x@2`, so it can occur only under some functions that bind `x` and `y`. For the purposes of this example, let us define `function1` that contains the above code as its function body:

```
let function1 = λ(x : Natural) → λ(y : Natural) →
  ( λ(x : Natural) → λ(y : Natural) → λ(x : Natural) → x + x@1 + x@2 ) y
```

Suppose we need to evaluate this expression (that is, evaluate "under a lambda"). In other words, we need to simplify the body of `function1` before applying that function.

To evaluate this expression correctly, we cannot simply substitute `y` instead of `x` in the body of the function. Note that:

- The outer `x` corresponds to `x@1` within the expression, so we need to substitute `y` instead of `x@1` while keeping `x` and `x@2` unchanged.
- Another `y` is already bound in an inner scope; so, we need to write `y@1` instead of `y`, in order to refer to the free variable `y` in the outside scope.
- When we remove the outer `x`, the free variable `x@2` will need to become `x@1`.

So, the result of evaluating the above expression must be:

```
λ(y : Natural) → λ(x : Natural) → x + y@1 + x@1
```

We can verify that Dhall actually performs this evaluation:

```
⊢ λ(x : Natural) → λ(y : Natural) → ( λ(x : Natural) → λ(y : Natural) → λ(x
    : Natural) → x + x@1 + x@2 ) y

λ(x : Natural) →
λ(y : Natural) →
λ(y : Natural) →
λ(x : Natural) →
  x + y@1 + x@1
```

1.4 Product types (records)

Product types are implemented via records. For example, `{ x = 123, y = True }` is a "record value" whose type is `{ x : Natural, y : Bool }`. Types of that form are called "record types".

There are no built-in tuple types, such as Haskell's and Scala's `(Int, String)`. Records with field names must be used instead. For instance, the (Haskell / Scala) tuple type `(Int, String)` may be translated into Dhall as the following record type: `{ _1 : Integer, _2 : Text }`. That record type has two fields named `_1` and `_2`. The two parts of the tuple may be accessed via those names and the "dot" operator:

```
⊢ :let tuple = { _1 = +123, _2 = "abc" }

tuple : { _1 : Integer, _2 : Text }

⊢ tuple._1

+123
```

Records can be nested: the record value `{ x = 1, y = { z = True, t = "abc" } }` has type `{ x : Natural, y : { z : Bool, t : Text } }`. Fields of nested record types may be accessed via the "dot" operator:

```
⊢ :let a = { x = 1, y = { z = True, t = "abc" } }

a : { x : Natural, y : { t : Text, z : Bool } }

⊢ a.y.t

"abc"
```

It is important that Dhall's records have **structural typing**: two record types are distinguished only via their field names and types, while record fields are unordered. So, the record types `{ x : Natural, y : Bool }` and `{ y : Bool, x : Natural }` are the same, while the types `{ x : Natural, y : Bool }` and `{ x : Text, y : Natural }` are different and unrelated to each other. There is no **nominal typing**: that is, no way of assigning a permanent unique name to a certain record type, as it is done in Haskell, Scala, and other languages in order to distinguish one record type from another.

For convenience, a Dhall program may define local names for types:

```
let RecordType1 = { a : Natural, b : Bool }
let x : RecordType1 = { a = 1, b = True }
let RecordType2 = { b : Bool, a : Natural }
let y : RecordType2 = { a = 2, b = False }
```

But the names `RecordType1` and `RecordType2` are no more than (locally defined) values that are used as type aliases. Dhall does not distinguish `RecordType1` and `RecordType2` from each other or from the literal type expression `{ a : Natural, b : Bool }`, as the order of record fields is not significant. So, the values `x` and `y` actually have the same type in this code.

It will be convenient to define a `Pair` type constructor:

```
let Pair = λ(a : Type) → λ(b : Type) → { _1 : a, _2 : b }
```

1.5 Co-product types ("union types")

Co-product types (called "union types" in Dhall) are implemented via tagged unions, for example: `< X : Natural | Y : Bool >`. Here `X` and `Y` are called the **constructors** of the given union type.

Values of union types are created via constructor functions. Constructor functions are written using the "dot" operator. For example, the Dhall expression `< X : Natural | Y : Bool >.X` is a function of type `Natural → < X : Natural | Y : Bool >`. Applying that function to a value of type `Natural` will create a value of the union type `< X : Natural | Y : Bool >`, as shown in this example:

```
let x : < X : Natural | Y : Bool > = < X : Natural | Y : Bool >.X 123
```

Constructor names are often capitalized (`X`, `Y`, etc.), but Dhall does not enforce that convention.

Constructors may have *at most one* argument. Constructors with multiple curried arguments (as in Haskell: `P1 Int Int | P2 Bool`) are *not* supported in Dhall. Record types must be used instead of multiple arguments. For example, Haskell's union type `P1 Int Int | P2 Bool` may be replaced by Dhall's union type `< P1 : { _1 : Integer, _2 : Integer } | P2 : Bool >`.

Union types can have empty constructors (that is, constructors without arguments). For example, the union type `< X : Natural | Y >` has an empty constructor (`Y`). The corresponding value is written as `< X : Natural | Y >.Y`. This is the only value that can be created via that constructor.

Union types can be nested, for example, `< T | X : < Y | Z : Natural > >`. Here is an (artificial) example of creating a value of that type:

```
let nested = < T | X : < Y | Z : Natural > >.X (< Y | Z : Natural >.Z 123)
```

It is important that Dhall's union types use **structural typing**: two union types are distinguished only via their constructor names and types, while constructors are unordered. So, the union types `< X : Natural | Y >` and `< Y | X : Natural >` are the same, while the types `< X : Natural | Y >` and `< X : Text | Y : Natural >` are different and unrelated to each other. There is no **nominal typing** for union types; that is, no way of assigning a permanent unique name to a certain union type, as it is done in Haskell, Scala, and other languages to distinguish one union type from another.

For convenience, Dhall programs often define local names for union types:

```
let MyXY = < X : Natural | Y : Bool >
let x : MyXY = MyXY.X 123
```

The constructor expression `MyXY.X` is a function of type `Bool → MyXY`. But the name `MyXY` is no more than a (locally defined) value that is used as a type alias. Dhall considers `MyXY` to be the same type as the literal type expressions `< X : Natural | Y : Bool >` and `< Y : Bool | X : Natural >`, as the order of a union type's constructors is not significant.

Dhall requires the union type's constructors to be explicitly annotated by the full union types. In Haskell or Scala, one may simply write `Left(x)` and `Right(y)` and the compiler will automatically find the relevant union type. But Dhall requires us to write `< Left : Text | Right : Bool >.Left x` or `< Left : Text | Right : Bool >.Right y`, fully specifying the union type whose value is being constructed.

An advantage of this syntax is that there is no need to keep the constructor names unique across all union types in scope (as it is necessary in Haskell and Scala). In Dhall, each union type may define arbitrary constructor names. For example, consider this code:

```
let Union1 = < Left : Text | Right >
let Union2 = < Left : Text | Right : Bool >
let u : Union1 = Union1.Left "abc"
let v : Union2 = Union2.Left "fgh"
let x : Union1 = Union1.Right
let y : Union2 = Union2.Right True
```

The types `Union1` and `Union2` are different because the constructors named `Right` require different data types within `Union1` and `Union2`. Constructor names are always written together with the union type. So, there is no conflict between the constructors `Union1.Left` and `Union2.Left`, or between `Union1.Right` and `Union2.Right`. (A conflict would occur if we could write simply `Left` and `Right` for those constructors, but Dhall does not support that.)

It will be convenient to define an `Either` type constructor:

```
let Either = λ(a : Type) → λ(b : Type) → < Left : a | Right : b >
```

1.6 Pattern matching

Pattern matching for union types is implemented via the `merge` keyword. Dhall's `merge` expressions are similar to `match/with` expressions in OCaml, `case/of` expressions in Haskell, and `match/case` expressions in Scala. One difference is that each case of a `merge` expression must specify an explicit function with a full type annotation.

As an example, consider a union type defined in Haskell by:

```
data P = X Int | Y Bool | Z    -- Haskell.
```

A function `toString` that prints values of type `P` can be written in Haskell via pattern matching:

```
-- Haskell
toString :: P -> String
toString x = case x of
  X x -> "X " ++ show x
  Y y -> "Y " ++ show y
  Z -> "Z"
```

The corresponding type is defined in Dhall by:

```
let P = < X : Integer | Y : Bool | Z >
```

Here is the Dhall code for a function that prints values of type `P`:

```
let Bool/show = https://prelude.dhall-lang.org/Bool/show
let pToText : P → Text = λ(x : P) →
  merge {
          X = λ(x : Integer) → "X " ++ Integer/show x,
          Y = λ(y : Bool) → "Y " ++ Bool/show y,
          Z = "Z",
        } x
```

The `merge` keyword works like a curried function whose first argument is a *record value* and the second argument is a value of a union type. The field names of the record must correspond to all the constructor names in the union type. The values inside the record are functions describing what to compute in each case where the union type's constructor has arguments. For no-argument constructors (e.g., for the constructor `Z` in the example shown above) the value inside the record does not need to be a function.

The second argument of `merge` is a value of a union type on which the pattern matching is being done.

Note that `merge` in Dhall is a special keyword, not a function, although its syntax (e.g., `merge { X = 0 } x`) looks like that of a curried function with two arguments. It is a syntax error to write `merge { X = 0 }` without specifying a value (`x`) of a union type.

1.7 The void type

The **void type** is a type that cannot have any values.

Dhall's empty union type (denoted by `<>`) is an example of a void type. Values of union types may be created only via constructors, but the union type `<>` has no constructors. So, no Dhall code will ever be able to create a value of type `<>`.

If a value of the void type existed, one would be able to compute from it a value of *any other type*. Although this may appear absurd, it is indeed an important property of the void type. This property can be expressed formally via the function that is often denoted by `absurd`. That function computes a value of an arbitrary type `A` given a value of the void type `<>`:

```
let absurd : <> → ∀(A : Type) → A
  = λ(x : <>) → λ(A : Type) → merge {=} x : A
```

This implementation depends on a special feature of Dhall: a `merge` expression with a type annotation. (The annotation `: A` in `merge {=} x : A` is for the entire `merge` expression, not for `x`.) This annotation makes the type checker accept an *empty* `merge` expression as a value of type `A`.

We need to keep in mind that the function `absurd` can never be actually applied to an argument value in any program, because one cannot construct any values of type `<>`. Nevertheless, the existence of the void type and a function of type `<> → ∀(A : Type) → A` is useful in some situations, as we will see below.

The type signature of `absurd` suggests a type equivalence between `<>` and the function type `∀(A : Type) → A`.

Indeed, the type `∀(A : Type) → A` is void. If we could have some expression `x` of that type, we would have then apply `x` to the void type and compute a value `x <>` of type `<>`. But that is impossible, as the type `<>` has no values.

So, the type `∀(A : Type) → A` can be used equally well to denote the void type.

One use case for the void type is to provide a "TODO" functionality. While writing Dhall code, we may want to leave a certain value temporarily unimplemented. However, we still need to satisfy Dhall's type checker and provide a value that appears to have the right type.

To achieve that, we rewrite our code as a function with an additional argument of the void type:

```
let our_program = λ(void : <>) → True
```

Now suppose we need a value `x` of some type `X` in our code, but we do not yet know how to implement that value. We write `let x : X = absurd void X` in the body of `our_program`:

```
let our_program = λ(void : <>) →
  let x : Integer = absurd void Integer
  in { x }    -- Whatever.
```

The typechecker will accept this code. Of course, we can never supply a value for the `void : <>` argument. So, our program cannot be evaluated until we replace the `absurd void X` by correct code computing a value of type `X`.

To shorten the code, define `let TODO = absurd void`. We can then write `TODO X` and pretend to obtain a value of any type `X`.

Note that the partially applied function `absurd void` is a value of type `∀(A : Type) → A`. So, we may directly require `TODO` as an argument of type `∀(A : Type) → A` in our program:

```
let our_program = λ(TODO : ∀(A : Type) → A) →
  let x = TODO Natural in { result = x + 123 }     -- Whatever.
```

We will be able to evaluate `our_program` only after replacing all `TODO` placeholders with suitable code.

1.8 The unit type

A **unit type** is a type that has only one distinct value.

Dhall's empty record type `{}` is a unit type. The type `{}` has only one value, written as `{=}`. This syntax denotes a record with no fields or values.

Another way of defining a unit type is via a union type with a single constructor, for example: `< One >` (or with any other name instead of "One"). The type `< One >` has a single distinct value, denoted in Dhall by `< One >.One`. In this way, one can define unit types with different names, when convenient.

Another equivalent definition of a unit type is via the following function type:

```
let UnitId = ∀(A : Type) → A → A
```

The only possible function of that type is the **polymorphic identity function**:

```
let identity : UnitId = λ(A : Type) → λ(x : A) → x
```

There is no other, inequivalent Dhall code that could implement a different function of that type. (This is a consequence of parametricity.) So, the type `UnitId` is a unit type because it has only one distinct value.

The unit type is denoted by `()` in Haskell, by `Unit` in Scala, by `unit` in OCaml, and similarly in other languages[4].

1.9 Type constructors

Type constructors in Dhall are written as functions from `Type` to `Type`.

In Haskell, one could define a type constructor as `type AAInt a = (a, a, Int)`. The analogous type constructor in Scala looks like `type AAint[A] = (A, A, Int)`. To encode this type constructor in Dhall, one writes an explicit function taking a parameter `a` of type `Type` and returning another type.

[4]https://en.wikipedia.org/wiki/Unit_type

Because Dhall does not have nameless tuples, we will use a record with field names `_1`, `_2`, and `_3` to represent a tuple with three parts:

```
let AAInt = λ(a : Type) → { _1 : a, _2 : a, _3 : Integer }
```

Then `AAInt` is a function that takes an arbitrary type (`a`) as its argument. The output of the function is the record type `{ _1 : a, _2 : a, _3 : Integer }`.

The type of `AAInt` itself is `Type → Type`. For more clarity, we may write that as a type annotation:

```
let AAInt : Type → Type = λ(a : Type) → { _1 : a, _2 : a, _3 : Integer }
```

Type constructors involving more than one type parameter are usually written as curried functions. Here is an example of defining a type constructor similar to Haskell's and Scala's `Either`:

```
let Either = λ(a : Type) → λ(b : Type) → < Left : a | Right : b >
```

Because Dhall does not have built-in tuple types, it is convenient to define a `Pair` type constructor:

```
let Pair = λ(a : Type) → λ(b : Type) → { _1 : a, _2 : b }
```

The types of `Either` and `Pair` is `Type → Type → Type`.

As with all Dhall types, type constructor names such as `AAInt`, `Either`, or `Pair` are just type aliases. Dhall distinguishes types and type constructors not by assigned names but by the type expressions themselves (**structural typing**).

1.10 The "Optional" type constructor

An `Optional` type (similar to Haskell's `Maybe` and Scala's `Option`) could be defined in Dhall like this:

```
let MyOptional = λ(a : Type) → < MyNone | MySome : a >
let x : MyOptional Natural = (MyOptional Natural).MySome 123
let y : MyOptional Text = (MyOptional Text).MyNone
```

The built-in `Optional` type constructor is a less verbose equivalent of this code. One writes `None Text` instead of `(MyOptional Text).MyNone`, and `Some 123` instead of `(MyOptional Natural).MySome 123`. (The type parameter `Natural` is determined automatically by Dhall.) Other than that, the built-in `Optional` type behaves as if it were a union type with constructor names `None` and `Some`.

Here is an example of using Dhall's `merge` for implementing a `getOrElse` function for `Optional` types:

```
let getOrElse : ∀(a : Type) → Optional a → a → a
  = λ(a : Type) → λ(oa : Optional a) → λ(default : a) →
    merge {
            None = default,
            Some = λ(x : a) → x
```

```
         } oa
```

The `Optional` type is a built-in Dhall type rather than a library-defined type for pragmatic reasons. First, a built-in type is integrated with the typechecker and supports more concise code (`Some 123` instead of `(Optional Natural).Some 123`). Second, the `Optional` type plays a special role when exporting data to JSON and YAML formats: record fields with `None` values are typically omitted from the generated configuration files.

1.11 Functions with type parameters

Functions with type parameters (also known as **generic functions**) are written as functions with extra arguments of type `Type`.

To see how this works, first consider a function that takes a pair of `Natural` numbers and swaps the order of numbers in the pair. We use a record type `{ _1 : Natural, _2 : Natural }` to represent a pair of `Natural` numbers. For brevity, we will define a type alias (`PairNatNat`) to denote that type:

```
let PairNatNat : Type = { _1 : Natural, _2 : Natural }
let swapNatNat : PairNatNat → PairNatNat = λ(p : PairNatNat) → { _1 = p._2,
   _2 = p._1 }
```

Note that the code of `swapNatNat` does not depend on having values of type `Natural`. The same logic would work with any two types. So, we can generalize `swapNatNat` to a function `swap` that supports arbitrary types of values in the pair. The two types will become type parameters; we will denote them by `a` and `b`. The input type of `swap` will be `{ _1 : a, _2 : b }` (which is the same as `Pair a b`) instead of `{ _1 : Natural, _2 : Natural }`, and the output type will be `{ _1 : b, _2 : a }` (which is the same as `Pair b a`). The type parameters are given as additional curried arguments of `swap`. The new code is:

```
let swap : ∀(a : Type) → ∀(b : Type) → Pair a b → Pair b a
  = λ(a : Type) → λ(b : Type) → λ(p : Pair a b) → { _1 = p._2, _2 = p._1 }
```

Note that some parts of the type signature of `swap` depend on the type parameters (namely, the types `Pair a b` and `Pair b a`). To be able to express that dependence, we need to specify the type parameter names in the type signature of `swap` as `∀(a : Type)` and `∀(b : Type)`.

Compare this with the type signature of `Pair`, which is written as `Type → Type → Type`. We could write, if we like, `Pair : ∀(a : Type) → ∀(b : Type) → Type`. That would be the same type signature in a longer syntax. But we cannot shorten the type signature of `swap` to `Type → Type → Pair a b → Pair b a`, because the names `a` and `b` would have become undefined. The type signature of `swap` requires a longer form that introduces the names `a` and `b`.

As another example, consider functions that extract the first or the second element of a pair. These functions also work in the same way for all types. So, it is useful to declare those functions as "generic functions" having type parameters:

```
let take_1 : ∀(a : Type) → ∀(b : Type) → Pair a b → a
  = λ(a : Type) → λ(b : Type) → λ(p : Pair a b) → p._1
let take_2 : ∀(a : Type) → ∀(b : Type) → Pair a b → b
  = λ(a : Type) → λ(b : Type) → λ(p : Pair a b) → p._2
```

A further example is the standard `map` function for `List`. The type signature of that function is `∀(a : Type) → ∀(b : Type) → (a → b) → List a → List b`.

When applying that function, the code must specify both type parameters (`a`, `b`):

```
let List/map = https://prelude.dhall-lang.org/List/map
in List/map Natural Natural (λ(x : Natural) → x + 1) [1, 2, 3]
  -- This is a complete program that returns [2, 3, 4].
```

A **polymorphic identity function** is written (with a full type annotation) as:

```
let identity : ∀(A : Type) → ∀(x : A) → A
  = λ(A : Type) → λ(x : A) → x
```

The type of the polymorphic identity function is of the form `∀(x : arg_t) → res_t` if we set `x = A`, `arg_t = Type`, and `res_t = ∀(x : A) → A`. Note that `res_t` is again a function type, and this time its result type (`A`) does not depend on the value of the argument (`x`). So, this type can be rewritten in the short form as `A → A`. We will usually write the identity function as:

```
let identity : ∀(A : Type) → A → A
  = λ(A : Type) → λ(x : A) → x
```

In Dhall, all function arguments (including all type parameters) must be introduced explicitly via the `λ` syntax. Each argument must have a type annotation, for example: `λ(x : Natural)`, `λ(a : Type)`, and so on.

However, a `let` binding does not require a type annotation. So, one may just write `let Pair = λ(a : Type) → λ(b : Type) → { _1 : a, _2 : b }`. The type of `Pair` will be determined automatically.

For complicated type signatures, it still helps to write type annotations, because Dhall will then detect some type errors earlier.

1.12 Modules and imports

Dhall has a simple file-based module system. The module system is built on the principle that each Dhall file must contain a valid program that is evaluated to a *single* result value. (Programs are often written in the form `let x = ... let y = ... in ...`, but the result is still a single value.)

A file's value may be imported into another Dhall file by specifying the path to the first Dhall file. The second Dhall file can then directly use that value as a sub-expression in any further code. For convenience, the imported value is usually assigned to a variable with a meaningful name.

In this way, each Dhall file is seen as a "module" that exports a single value. There are no special keywords to denote the exported value; there is only one such value in any case.

Here is an example: the first file contains a list of numbers, and the second file computes the sum of those numbers.

```
-- This file is `./first.dhall`.
[1, 2, 3, 4]
```

```
-- This file is `./sum.dhall`.
let input_list = ./first.dhall  -- Import from a file at a relative path.
let sum = https://prelude.dhall-lang.org/Natural/sum  -- Import from URL.
in sum input_list
```

Running `dhall` on the second file will compute and show the result:

```
$ dhall --file ./sum.dhall
10
```

Although each Dhall module exports only one value, that value may be a record with many fields. Record fields may contain values and/or types. In that way, Dhall modules may export a number of values and/or types:

```
-- This file is `./SimpleModule.dhall`.
let UserName = Text
let UserId = Natural
let printUser = λ(name : UserName) → λ(id : UserId) → "User:
    ${name}[${Natural/show id}]"

let validate : Bool = ./NeedToValidate.dhall -- Import that value from another
    module.
let test = assert : validate ≡ True   -- Cannot import this module unless
    `validate` is `True`.

in {
  UserName,
  UserId,
  printUser,
}
```

When this Dhall file is evaluated, the result is a record of type `{ UserName : Type, UserId : Type, printUser : Text → Natural → Text }`. We could say that this module exports two types (`UserName`, `UserId`) and a function `printUser`. But technically it just exports a single value (a record). Exporting a record is a common technique in Dhall libraries.

This module may be imported in another Dhall program like this:

```
let S = ./SimpleModule.dhall -- Just call it `S` for short.
let name : S.UserName = "first_user"
let id : S.UserId = 1001
let printed : Text = S.printUser name id
-- Continue writing code.
```

Here we used the types and the values exported from `SimpleModule.dhall`. This code will not compile unless all types match, including the imported values.

All fields of a Dhall record are always public. To make values in a Dhall module private, we simply do not include those values into the final exported record. Local values declared using `let x = ...` inside a Dhall module will not be exported (unless they are included in the final exported record).

In the example just shown, the file `SimpleModule.dhall` defines the local values `test` and `validate`. Those values are typechecked and computed inside the module, but are not exported. In this way, sanity checks or unit tests included within a module will be validated but will remain invisible to other modules.

Other than importing values from files, Dhall supports importing values from Web URLs and from environment variables. Here is an example of importing the Dhall list value `[1, 1, 1]` from an environment variable called `XS`:

```
$ echo "let xs = env:XS in List/length Natural xs" | XS="[1, 1, 1]" dhall
3
```

In this way, Dhall programs may perform computations with external inputs.

However, most often the imported Dhall values are not simple data but records containing types, values, and functions.

Dhall denotes imports via special syntax:

- If a Dhall value begins with `/`, it is an import of a file from an absolute path.
- If a Dhall value begins with `./`, it is an import of a file from a relative path (relative to the directory containing the currently evaluated Dhall file).
- If a Dhall value begins with `http://` or `https://`, it is an import from a Web URL.
- A Dhall value of the form `env:XYZ` is an import from a shell environment variable `XYZ` (in Bash, this would be `$XYZ`). It is important to use no spaces around the `:` character, because `env : XYZ` means a value `env` of type `XYZ`.

It is important that the import paths, environment variable names, and SHA256 hash values are *not strings*. They are hard-coded and cannot be manipulated at run time.

It is not possible to import a file whose name is computed by concatenating some strings. For instance, one cannot write anything like `let x = ./Dir/${filename}`, with the intention of substituting the value `filename` as part of the path to the imported file.

The contents of the imported resource my be treated as plain text or as binary data, instead of treating it as Dhall code. This is achieved with the syntax `as Text` or `as Bytes`.

For example, environment variables typically contain plain text rather than Dhall code. So, they should be imported `as Text`:

```
⊢ env:SHELL as Text

"/bin/bash"
```

But one cannot read an external resource as a string and then use that string as a URL or file path for another import.

Instead, it is possible to read the *path* to an imported resource as a value of a special type. The syntax `as Location` enables that option:

```
⊢ env:SHELL as Location

< Environment : Text | Local : Text | Missing | Remote : Text >.Environment
   "SHELL"
```

The result is a value of a union type that describes all supported external resources.

However, `Location` values cannot be reused to perform further imports.

Apart from requiring hard-coded import paths, Dhall imposes other limitations on what can be imported to help users maintain a number of security guarantees:

- All imported modules are required to be valid (must pass typechecking and optionally SHA256 hash matching).

- All imported resources are loaded and validated at typechecking time, before any evaluation may start.

- Circular imports are not allowed: no resource may import itself, either directly or via other imports.

- Imported values are referentially transparent: a repeated import of the same external resource is guaranteed to give the same value (if the import is successful).

- The authentication headers must be either hard-coded or imported; they cannot be computed at evaluation time.

- CORS restrictions are implemented, so Web URL imports that require authentication headers will not leak those headers to other Web servers.

See the Dhall documentation on safety guarantees[5] for more details.

[5] https://docs.dhall-lang.org/discussions/Safety-guarantees.html

1.12.1 Organizing modules and submodules

The Dhall standard library (the "prelude"[6]) stores code in subdirectories organized by type name. For instance, functions working with the `Natural` type are in the `Natural/` subdirectory, functions working with lists are in the `List/` subdirectory, and so on. This convention helps make the code for imports more visual:

```
let Integer/add = https://prelude.dhall-lang.org/Integer/add
let List/concatMap = https://prelude.dhall-lang.org/List/concatMap
let Natural/greaterThan = https://prelude.dhall-lang.org/Natural/greaterThan
let Natural/lessThanEqual = https://prelude.dhall-lang.org/Natural/lessThanEqual
let Optional/default = https://prelude.dhall-lang.org/Optional/default
let Optional/map = https://prelude.dhall-lang.org/Optional/map     -- And so on.
```

The import mechanism can be used as a module system that supports creating libraries of reusable code. For example, suppose we put some Dhall code into files named `./Dir1/file1.dhall` and `./Dir1/file2.dhall`. We can then import those files like this:

```
let Dir1/file1 = ./Dir1/file1.dhall
let Dir1/file2 = ./Dir1/file2.dhall
in ???
```

Code in `file1.dhall` could also import `file2.dhall` using a relative path, for example like this: `let x = ./file2.dhall`.

Keep in mind that the names such as `Integer/add` or `Dir1/file1` are just a visually helpful convention. The fact that both files `file1.dhall` and `file2.dhall` are located in the same subdirectory (`Dir1`) has no special significance. Any file can import any other file, as long as an import path (absolute or relative) is given.

An import such as `let Dir1/file1 = ./Dir1/file1.dhall` might appear to suggest that `file1` is a submodule of `Dir1` in some sense. But actually Dhall treats all imports in the same way regardless of path; it does not have any special concept of "submodules". The file `./Dir1/file1.dhall` is treated as a module like any other.

Names like `Dir1/file` are often used in Dhall libraries, but Dhall does not give any special treatment to such names. Dhall will neither require nor verify that `let Dir1/file1 = ...` imports a file called `file1` in a subdirectory called `Dir1`.

To create a hierarchical library structure of modules and submodules, the Dhall standard library uses nested records. Each module has a top-level file called `package.dhall` that defines a record with all values from that module. Some of those values could be again records containing values from other modules (that also define their own `package.dhall` in turn). The top level of Dhall's standard prelude is a file called `[package.dhall](https://prelude.dhall-lang.org/package.dhall)` that contains a record with all modules in the prelude. A Dhall file may import the entire prelude and access its submodules like this:

```
let p = https://prelude.dhall-lang.org/package.dhall -- Takes a while to import!
```

[6] https://prelude.dhall-lang.org

```
let x = p.Bool.not (p.Natural.greaterThan 1 2)     -- We can use any module now.
  in ???
```

The standard prelude is not treated specially by Dhall. It is just an ordinary import from a Web URL. A user's own libraries and modules may have a similar structure of nested records and may be imported as external resources in the same way. In this way, users can organize their Dhall configuration files and supporting functions via shared libraries and modules.

1.12.2 Frozen imports and semantic hashing

Importing from external resources (files, Web URLs, or environment variables) is a form of a side effect because the contents of those resources may change at any time. Dhall has a feature called **frozen imports** for ensuring that the contents of an external resource does not change unexpectedly. Frozen imports are annotated by the SHA256 hash value of the imported content's normal form after a full evaluation.

As an example, create a file called `simple.dhall` containing just the number `3`:

```
-- This file is ‘simple.dhall‘.
3
```

That file may be imported via the following frozen import:

```
-- This file is ‘another.dhall‘.
./simple.dhall
    sha256:15f52ecf91c94c1baac02d5a4964b2ed8fa401641a2c8a95e8306ec7c1e3b8d2
```

This import expression is annotated by the SHA256 hash value corresponding to the Dhall expression `3`. If the user modifies the file `simple.dhall` so that it evaluates to anything other than `3`, the hash value will become different and the frozen import will fail.

A frozen import is guaranteed to produce the same value every time, because the imported value's hash is always validated. If the contents of the external resource changes and its SHA256 hash no longer matches the annotation, Dhall will raise an error.

Hash values are computed from the *normal form* of Dhall expressions, and the normal forms are computed only after successful typechecking. For this reason, the semantic hash of a Dhall program remains unchanged under any valid refactoring. For instance, we may add or remove comments; reformat the file with fewer or with more spaces or empty lines; change the order of fields in records or the order of constructors in union types; rename, add, or remove local variables; and even change import URLs (as long as the imported content remains equivalent). The hash value will remain the same as long as the normal form of the final evaluated expression remains the same. This form of hashing is known as **semantic hashing**.

The Dhall interpreter will cache all frozen imports in the local filesystem, using the SHA256 semantic hash value as part of the file name. This makes importing libraries faster after the first time.

Keep in mind that Dhall programs with non-frozen imports may produce different results when evaluated at different times. An example of that behavior is found in Dhall's test suite. It imports a randomness source[7], which is a Web service that returns a new random string each time it is called. So, this Dhall program:

```
https://test.dhall-lang.org/random-string as Text -- This is a complete program.
```

will return a different result *each time* it is evaluated:

```
$ echo "https://test.dhall-lang.org/random-string as Text" | dhall
''
Gajnrpgc4cHWeoYEUaDvAx5qOHPxzSmy
''
$ echo "https://test.dhall-lang.org/random-string as Text" | dhall
''
tH8kPRKgH3vgbjbRaUYPQwSiaIsfaDYT
''
```

One cannot guarantee that any given Web URL always returns the same results (or always remains reachable). To ensure that all expressions are referentially transparent, each successful import is cached on its first use within a Dhall program. If a Dhall program imports the same external resource several times, all those imports will always have the same value when the program is evaluated. For example, if a program imports `https://test.dhall-lang.org/random-string` several times, the first imported value will be internally cached and substituted for all subsequent imports of that resource.

For this reason, a Dhall program cannot query a Web URL several times and make decisions based on the changes in its response.

1.12.3 Import alternatives

Dhall has features for providing alternative external resources to be used when an import fails. Any number of alternative imports may be specified, separated by the question mark operator (`?`):

```
let Natural/lessThan = ./MyLessThanImplementation.dhall
  ? ./AnotherImplementationOfLessThan.dhall
  ? https://prelude.dhall-lang.org/Natural/lessThan
```

This mechanism resolves only "non-fatal" import failures: that is, failures to read an external resource. A "fatal" import failure means that the external resource was available but gave a Dhall expression that failed to parse, to typecheck, to vali-

[7] https://test.dhall-lang.org/random-string

date the given semantic hash, or violated some import restrictions (e.g., a circular import).

The operator for alternative imports (`?`) is designed for situations where the same Dhall resource might be stored in different files or at different URLs, some of which might be temporarily unavailable. If all alternatives fail to read, the import fails (and the entire Dhall program fails to type-check).

Other than providing import alternatives, Dhall does not support any possibility of reacting to an import failure in a custom way. The intention is to prevent Dhall programs from depending on side effects due to timing issues or network availability. Dhall programs are intended to be pure and referentially transparent values even in the presence of imports.

The special keyword `missing` denotes an external resource that will *never* be available. It works as a neutral element of the `?` operation: `x ? missing` is the same as `x`.

The `missing` keyword is sometimes used as a trick to speed up import loading for frozen imports. To achieve that, annotate a `missing` import with an SHA256 hash value and provide an alternative:

```
let Natural/lessThan
  = missing
    sha256:3381b66749290769badf8855d8a3f4af62e8de52d1364d838a9d1e20c94fa70c
  ? https://prelude.dhall-lang.org/Natural/lessThan
```

If the function `Natural/lessThan` has been already cached, it will be retrieved from the cache without resolving any URLs. Otherwise, there will be a URL lookup and the function will be loaded and cached on the local machine's filesystem. The next time this Dhall program is run, there will be no URL lookups.

1.13 Miscellaneous features

Multiple `let x = y in z` bindings may be written next to each other without writing "`in`". For example:

```
let a = 1
let b = 2
in a + b  -- This is a complete program that evaluates to 3.
```

Because of this syntax, Dhall programs often have the form `let ... let ... let ... in ...`, where the "`in`" occurs only at the end. This book writes most snippets of Dhall code in the form `let a = ...` without the trailing `in`. It is implied that all those `let` declarations are part of a larger program with "`in`" somewhere at the end.

When we are working with the Dhall interpreter, we may write a standalone `let` declaration. The syntax is `:let`.

For instance, we may define the type constructor `Pair` shown above:

```
$ dhall repl
```

```
Welcome to the Dhall v1.42.2 REPL! Type :help for more information.
⊢ :let Pair = λ(a : Type) → λ(b : Type) → { _1 : a, _2 : b }

Pair : ∀(a : Type) → ∀(b : Type) → Type
```

Dhall does not require capitalizing the names of types and type parameters. In this book, we capitalize all type constructors (such as `List`). Simple type parameters are usually not capitalized in Dhall libraries (`a`, `b`, etc.), but we will sometimes write capitalized type parameters (`A`, `B`, etc.) for additional clarity. Values are never capitalized in this book.

1.13.1 Almost no type inference

Dhall has almost no type inference. The only exception are the `let` bindings, such as `let x = 1 in ...`, where the type annotation for `x` may be omitted. Other than in `let` bindings, the types of all bound variables must be written explicitly.

Although this makes Dhall programs more verbose, it also removes the "magic" from the syntax of certain idioms in functional programming. In particular, Dhall requires us to write out all type parameters and all type quantifiers, to choose carefully between `∀(x : A)` and `λ(x : A)`, and to write type annotations for *types* (such as, `F : Type → Type`). This verbosity has helped the author in learning some of the more advanced concepts of functional programming.

1.13.2 Strict and lazy evaluation. Partial functions

In a programming language, the **strict evaluation** strategy means that all subexpressions in a program are evaluated even if they are not used to compute the final result of the program. The **lazy evaluation** strategy means that subexpressions are evaluated only if they are needed for computing the program's final result.

In this sense, the Dhall interpreter almost always uses lazy evaluation. For example, it will be quick to run Dhall programs similar to this:

```
let x = some_function 1000000 ??? -- Imagine that this is a long computation.
in 123
```

The value `x` will not be evaluated because it is not actually needed for computing the final value (`123`).

On the other hand, if we modify this program to include an `assert` test (that feature will be described below), the program will take a longer time to run:

```
let x = some_function 1000000 ??? -- Imagine that this is a long computation.
let _ = assert : validate x ≡ True -- Validate the result in some way.
in 123
```

The value `x` now needs to be evaluated in order to verify that `validate x` actually returns `True`.

Validation of `assert` expressions will happen at typechecking time, and Dhall's *typechecking* is strict (not lazy). For this reason, the value `x` will get evaluated in the program shown above, even though `x` is not used anywhere later in the code. Also, a type error such as `let x : Natural = "abc"` will prevent the entire program from evaluating, even if the ill-typed value `x` is never used in any expressions later in the program.

Another case where Dhall enforces strict evaluation is when importing values from external resources. Each imported value is loaded and validated at typechecking time, even if the value is not used in the code that follows:

```
let constZero = λ(x : Natural) → 0   -- This function ignores its argument.
let ??? = constZero ./nonexisting_file.dhall -- Error at typechecking time!
```

The typechecking stage is analogous to the compile-time stage in compiled programming languages. At that stage, the Dhall interpreter resolves all imports and then typechecks all sub-expressions in the program, whether they are used or not. (Imports must be resolved first, in order to be able to proceed with typechecking.)

When no type errors are found, the interpreter goes on evaluating the program to a normal form, using the lazy evaluation strategy.

It is important to keep in mind that in almost all cases a well-typed Dhall program should give the same result whether evaluated lazily or strictly.

The lazy and strict evaluation strategies will give different results in two situations:

- When a certain sub-expression creates a lazy side effect whose execution may influence the result value.
- When a certain "rogue" sub-expression *cannot* be evaluated (because it will either crash the program or enter an infinite loop).

To implement the first case, we would need to create a Dhall expression containing a side effect. The only side effect in Dhall is importing an external resource. However, Dhall makes all imports strictly evaluated and validated at typechecking time. So, imports are never lazily evaluated. As Dhall has no other side effects, we see that the first case does not create a difference between lazy and strict evaluation strategies at evaluation time.

To see how the second case could work, suppose a program defines a rogue expression but does it in such a way that the rogue expression is *not* actually needed for computing the final result. Under the lazy evaluation strategy, the rogue expression will not be evaluated (because it is not used), and the program will complete successfully. But the strict evaluation strategy will try to compute all expressions (whether or not they are used for obtaining the final result). In that case, the program will fail due to failure evaluating the rogue expression.

Dhall goes quite far towards making rogue expressions impossible. This is due to Dhall's specific choice of features and strict typechecking:

- A function call will typecheck only if all arguments have correct types.
- A pattern-matching expression will typecheck only if it handles *all* parts of the union type being matched.
- All `if / then` constructions must have an `else` clause; both clauses must be values of the same type.
- There are no "undefined" values: Dhall has no analog of Haskell's "bottom" or of Java's "null" or of Scala's `???`.
- A program cannot create exceptions or other run-time errors.
- Infinite loops are not possible; every loop must have an upper bound on the number of iterations.
- All lists must have an upper bound on length.

These features eliminate large classes of programmer errors that could create rogue expressions inadvertently. Nevertheless, two possibilities for creating rogue expressions still remain:

- Create such a large data structure that no realistic computer could fit it in memory.
- Start a computation that takes such a long time that no realistic user could wait for its completion.

These situations are impossible to avoid, because (even with Dhall's restrictions) it is not possible to determine in advance the maximum memory requirements and run times of an arbitrary given program.

Here are examples of implementing these two possibilities in Dhall:

```
let doubleText = λ(x : Text) → x ++ x
-- It is important that Dhall uses lazy evaluation here.
-- A strict evaluation of ‘petabyte‘ would require over a petabyte of memory!
let petabyte : Text = Natural/fold 50 Text doubleText "x"

-- A strict evaluation of ‘slow‘ would require over a thousand years!
let slow : Natural = Natural/fold 10000000000000000 Natural (λ(x : Natural) →
    Natural/subtract x 1) 1
```

Triggering one of these situations will implement (an artificial example of) a **partial function** in Dhall. A function is partial if it works only for a certain subset of possible argument values; when applied to other values, the partial function will crash or will never complete evaluation. Typically, a partial function applied to wrong values will create a rogue expression.

As an example, consider the following code that implements a partial function called `crash`. Evaluating `crash False` returns an empty string. But `crash True` tries to return a **petabyte** of text, which will almost certainly crash any computer:

```
let crash = λ(b : Bool) → if b then petabyte else ""
let _ = assert : crash False ≡ ""   -- OK.
-- let _ = assert : crash True ≡ ""  -- This will crash!
```

Barring such artificial situations, rogue expressions and partial functions are impossible to implement in Dhall.

So, a Dhall programmer typically does not need to distinguish strict and lazy evaluation. One can equally well imagine that all Dhall expressions are lazily evaluated, or that they are all strictly evaluated. The result values of any Dhall program will be the same, as long as memory or time constraints are satisfied, and as long as all imported external resources are valid.

1.13.3 No computations with custom data

In Dhall, most built-in types (`Double`, `Bytes`, `Date`, `Time`, `TimeZone`) are completely opaque to the user. One can specify literal values of those types, and the only operation available for them is printing their values as `Text` strings. Those types are intended for creating strongly-typed configuration data schemas and for safely exporting data to configuration files.

The built-in types `Bool`, `List`, `Natural`, and `Text` support more operations. In addition to specifying literal values of those types and printing them to `Text`, a Dhall program can:

- Use `Bool` values as conditions in `if` expressions or with the Boolean operations (`&&`, `||`, `==`).
- Add, multiply, and compare `Natural` numbers.
- Concatenate lists and compute certain other functions on lists (`List/indexed`, `List/length`).
- Concatenate, interpolate, and replace substrings in `Text` strings.

Dhall cannot compare `Text` strings for equality or compute the length of a `Text` string. Neither can Dhall compare `Double` values or date/time values with each other. Comparison functions are only possible for `Bool`, `Integer`, and `Natural` values.

1.13.4 No general recursion

Unlike most other programming languages, Dhall does not support recursive definitions, neither for types nor for values. The only recursive data structure directly supported by Dhall is the built-in type `List` representing finite sequences. The only way to write a loop in Dhall is to use the built-in functions `List/fold`, `Natural/fold`, and "fold-like" functions derived from them, such as `List/map` and others. Loops written in that way are statically guaranteed to terminate because the total number of iterations is always fixed in advance.

Unlike other programming languages, Dhall does not support "while" or "until" loops. Those loops perform as many iterations as needed to reach a given stopping condition. In certain cases, it is not possible to find out in advance whether the stopping condition will ever be reached. So, programs that contain "while" or "until" loops are not statically guaranteed to terminate.

A list may be created only if the required length of the list is known in advance. It is not possible to write a program that creates a list by adding more and more elements until some condition is reached, without setting an upper limit in advance.

Similarly, all text strings are statically limited in length.

Although Dhall does not support recursion directly, one can use certain tricks (the Church encoding and existential types) to write non-recursive definitions that simulate recursive types, recursive functions, and "lazy infinite" data structures. Later chapters in this book will show how that can be achieved.

In practice, this means Dhall programs are limited to working with finite data structures and fold-like iterative functions on them. Recursive code can be translated into Dhall only if there is a termination guarantee: the total number of nested recursive calls must be bounded in advance. Within these limitations, Dhall supports a wide variety of iterative and recursive computations.

1.13.5 No side effects

Dhall is a purely functional language with no side effects. All values are referentially transparent. There are no mutable values, no exceptions, no multithreading, no writing to files, no printing, etc.

A valid Dhall program is a single expression that will be evaluated to a normal form by the Dhall interpreter. The user may then print the result to the terminal, or convert it to JSON, YAML, and other formats. But that happens outside the Dhall program.

The only feature of Dhall that is in some way similar to a side effect is the "import" feature: a Dhall program can read Dhall values from external resources (files, Web URLs, and environment variables). The result of evaluating a Dhall program will then depend on the contents of the external resources.

However, this dependency is invisible inside Dhall code, where each import looks like a constant value. This is because Dhall implements one-time, cached, and read-only imports. Dhall reads import values similarly to the way a mathematical function reads its arguments. The paths to external resources must be hard-coded; they are not strings and cannot be computed at run time. Even though the actual external resource may change during evaluation, the Dhall interpreter will ignore those changes and use only the first value obtained by reading the resource. A repeated import of the same resource will always give the same Dhall value. So, it is not possible to write a Dhall program that repeatedly reads a value from an external file and reacts in some way to changes in the file's contents.

A Dhall program also cannot have a custom behavior reacting to a failure *while importing* the external resources. There is only a simple mechanism providing fall-back alternative imports in case a resource is missing. If a resource fails to read despite fall-backs, or fails to typecheck, or fails the integrity check, the Dhall interpreter will fail the entire program.

Most often, imports are used to read library modules with known contents that is not expected to change. Dhall supports that use case with its "frozen imports" feature. If all imports are frozen, a Dhall program is guaranteed to give the same results each time it is evaluated.

1.13.6 Guaranteed termination

In System Fω, all well-typed expressions are guaranteed to evaluate to a unique final result. Thanks to this property, the Dhall interpreter is able to guarantee that any well-typed Dhall program will be evaluated in finite time to a unique **normal form** expression (that is, to an expression that cannot be simplified any further).

Evaluation of a well-typed Dhall program will never create infinite loops or throw exceptions due to missing or invalid values or wrong types at run time, as it happens in other programming languages. It is guaranteed that the correct normal form will be computed, given enough time and computer memory.

Invalid Dhall programs will be rejected at the typechecking phase. The typechecking itself is also guaranteed to complete within finite time.

The price for those termination and safety guarantees is that the Dhall language is *not* Turing-complete. A Turing-complete language must support programs that do not terminate as well as programs for which it is not known whether they terminate. Dhall supports neither general recursion nor `while`-loops nor any other iterative constructions whose termination is not assured. Iterative computations are possible in Dhall only if the program specifies the maximum number of iterations in advance.

The lack of Turing-completeness is not a significant limitation for a wide scope of Dhall usage, as this book will show. Dhall can still perform iterative or recursive processing of numerical data, lists, trees, or other user-defined recursive data structures.

The termination guarantee does *not* mean that Dhall programs could never exhaust the memory or could never take too long to evaluate. As Dhall supports arbitrary-precision integers, it is possible to write a Dhall program that runs a loop with an extremely large number of iterations. It is also possible to come up with short Dhall expressions creating data structures that consume terabytes or petabytes of memory. We will see some examples of such "rogue expressions" later in this book.

However, it is improbable that a programmer creates a rogue expression by mistake while implementing ordinary tasks in Dhall. A malicious adversary could try to inject such expressions into Dhall programs. To mitigate that possibility, Dhall implements strict guardrails on external imports.

1.14 Overview of the standard library ("prelude")

Dhall's standard library[8] has a number of modules containing utility functions.

The library functions are mostly organized in subdirectories whose names indicate the main type with which the functions work. So, functions in the `Bool` subdirectory are for working with the `Bool` type, functions in the `List` subdirectory work with lists, and so on.

All built-in Dhall functions have the corresponding standard library functions (just for the purpose of clean re-export). For example, the built-in function `Date/show` has the corresponding standard library code exported at `https://prelude.dhall-lang.org/Da` (which just calls the built-in function).

In addition, there are subdirectories such as `Function` and `Operator` that do not correspond to a specific type but are collections of general-purpose utilities.

Library functions for working with `Natural` and `Integer` numbers include functions such as `Natural/lessThan`, `Integer/lessThan`, and so on. Functions for working with lists include `List/map`, `List/filter`, and other utility functions.

All those functions are implemented through Dhall built-ins. As Dhall intentionally includes only a small number of built-in functions, implementing functionality such as `List/take` and `List/drop` results in slower and/or larger code than one might expect.

TODO:describe DirectoryFile, JSON, Function, Operator

[8] https://prelude.dhall-lang.org

2 Other features of Dhall's type system

2.1 Working with records polymorphically

"Polymorphic records" is a feature of some programming languages where, say, a record of type `{ x : Natural, y : Bool }` is considered to be a subtype of the record type `{ y : Bool }`. A function that requires its argument to have type `{ y : Bool }` will then also accept an argument of type `{ x : Natural, y : Bool }`. (The value `x` will be simply ignored.) In effect, languages with polymorphic records will interpret the record type `{ y : Bool }` as the type of any record having a Boolean field `y` and possibly other fields that the code (in this scope) does not need to know about.

Dhall does not supports subtyping or polymorphic records, but it does include some facilities to work with records polymorphically.

A typical use case for polymorphic records is when a function requires an argument of a record type `{ a : A, b : B }`, but we would like that function to accept records with more fields, for example, of type `{ a : A, b : B, c : C, d : D }`. The function only needs the fields `a` and `b` and should ignore any other fields the record may have.

To implement this behavior in Dhall, use a field selection operation: any unexpected fields will be automatically removed from the record.

```
let MyTuple = { _1 : Bool, _2 : Natural}
let f = λ(tuple : MyTuple) → tuple._2
let r1 = { _1 = True, _2 = 123, _3 = "abc", other = [ 1, 2, 3 ] }
in f r1.(MyTuple)  -- This is a complete program that returns 123.
```

The field selection operation `r1.(MyTuple)` removes all fields other than those defined in the type `MyTuple`. We cannot write `f r1` because `r1` does not have the type `MyTuple`. Instead, we write `f r1.(MyTuple)`. We would need to use the field selection each time we call the function `f`.

Another often needed feature is providing default values for missing fields. This is implemented with Dhall's record update operation (`//`):

```
let MyTuple = { _1 : Bool, _2 : Natural}
let myTupleDefault = { _1 = False, _2 = 0 }
let f = λ(tuple : MyTuple) → tuple._2
let r2 = { _2 = 123, _3 = "abc", other = [ 1, 2, 3 ] }
```

```
in f (myTupleDefault // r2).(MyTuple)  -- This is a complete program that
    returns 123.
```

We cannot write `f r2.(MyTuple)` because `r2` does not have the required field `_1`. The default record `myTupleDefault` provides that value.

The expression `(myTupleDefault // r).(MyTuple)` will accept record values `r` of any record type whatsoever. If `r` contains fields named `_1` and/or `_2`, the expression `myTupleDefault // r` will preserve those fields while filling in the default values for any missing fields. The field selection `.(MyTuple)` will remove any other fields.

The built-in Dhall operations `//` and `.()` can be viewed as functions that accept polymorphic record types. For instance, `r.(MyTuple)` will accept records `r` having the fields `_1 : Bool , _2 : Natural` and possibly any other fields. Similarly, `myTupleDefault // r` will accept records `r` of any record type and return a record that is guaranteed to have the fields `_1 : Bool` and `_2 : NAtural`.

But Dhall cannot directly describe the type of records with unknown fields. (There is no type that means "any record".) So, one cannot write a Dhall function taking an arbitrary record `r` and returning `r.(MyTuple)` or `myTupleDefault // r`.

Dhall programs must write expressions such as `myTupleDefault // r` or `r.(MyTuple)` at each place (at call site) where record polymorphism is required.

2.2 Types and values

In Dhall, as in every programming language, types are different from values. Each value has an assigned type, but it is not true that each type has only one assigned value.

Dhall will check that each value in a program has the correct type and that all types match whenever functions are applied to arguments, or when explicit type annotations are given.

Other than that, Dhall treats types and values in a largely similar way. Types may be assigned to new named values, stored in records, and passed as function parameters using the same syntax as when working with values.

For instance, we may write `let x : Bool = True` to define a variable of type `Bool`. Here, we used the type `Bool` as a type annotation for the variable `x`. But we may also write `let y = Bool` to define a variable `y` whose value is the type symbol `Bool` itself. Then we will be able to use `y` in type annotations, such as `x : y`. The type of `y` itself will be `Type`.

To find out the type of an expression, one can write `:type` in the Dhall interpreter:

```
$ dhall repl
Welcome to the Dhall v1.42.2 REPL! Type :help for more information.
⊢ :type True

Bool
```

```
⊢ :type Bool

Type
```

Dhall defines functions with the λ syntax:

```
let inc = λ(t : Natural) → t + 1
```

The same syntax works if `t` were a type parameter (having type `Type`):

```
let f = λ(t : Type) → λ(x : t) → { first = x, second = x }
```

Records and unions may use types or values as their data, as in these (artificial) examples:

```
⊢ :type { a = 1, b = Bool, c = "xyz" }

{ a : Natural, b : Type, c : Text }

⊢ :type < A : Type | B : Text >.A Natural

< A : Type | B : Text >
```

The built-in type constructors `List` and `Optional` are limited to *values*; one cannot create a `List` of types in the same way as one creates a list of integers, and one cannot use `Optional` with types.

```
⊢ :let a = [ 1, 2, 3 ]

a : List Natural

⊢ :let b = [ Bool, Natural, Text ]

Error: Invalid type for <List>

⊢ :let c = Some Natural

Error: <Some> argument has the wrong type
```

Nevertheless, Dhall can implement a data structure similar to an `Optional` containing a type.

```
let OptionalT = < None | Some : Type >
```

Later chapters in this book will show how to define type-level data structures such as a list of type symbols.

2.3 Dependent types in Dhall

Dependent types are types that depend on *values*.

Dhall supports **dependent functions**: those are functions whose output type depends on the input value. More generally, Dhall allows an argument type to depend on any previously given curried arguments.

A simple instance of this dependence is the type of the polymorphic identity function:

```
let example1 : Type = ∀(A : Type) → ∀(x : A) → A
```

Here the second curried argument (`x : A`) is designated as having the type (`A`) given by the first curried argument (`A : Type`). So, the type of the second argument depends on the first argument.

As another example, consider the following function type:

```
let example2 : Type = ∀(F : Type → Type) → ∀(A : Type) → ∀(x : A) → F A
```

In the type `example2`, the argument `x` has type `A`, which is given by a previous argument. The output type `F A` depends on the first two arguments.

In Dhall, one can define functions from types to types or from values to types via the same syntax as for defining ordinary functions. As an example, look at this function that transforms a value into a type:

```
let f : ∀(x : Bool) → Type  -- From value to type.
  = λ(x : Bool) → if x then Natural else Text
```

The result of evaluating `f False` is the *type* `Text` itself. This is an example of a **dependent type**, that is, a type that depends on a value (`x`). This `f` can be used within the type signature for another function as a type annotation, like this:

```
let some_func_type = ∀(x : Bool) → ∀(y : f x) → Text
```

A value of type `some_func_type` is a curried function that takes a natural number `x` and a second argument `y`. The type of `y` must be either `Natural` or `Text` depending on the *value* of the argument `x`.

```
let some_func : some_func_type = λ(x : Bool) → λ(y : f x) → "abc" -- Whatever.
```

If we imagine uncurrying that function, we would get a function of type that we could write symbolically as `{ x : Bool, y : f x } → Text`. This type is not valid in Dhall, because a record's field types must be fixed and cannot depend on the *value* of another field. Such "dependent records" or "dependent pairs" are directly supported in more advanced languages that are intended for working with dependent types. We will show later in this book how Dhall can encode dependent pairs despite that limitation.

The type `∀(x : Bool) → f x` is also a form of a dependent type, known as a "dependent function".

Dhall's implementation of dependent types is limited to the simplest use cases. The main limitation is that Dhall cannot correctly infer types that depend on values in `if/then/else` expressions or in pattern-matching expressions.

The following example (using the function `f` defined above) shows that Dhall does not track correctly the dependent types inside an `if` branch:

```
⊢ :let g : ∀(x : Bool) → f x → Text = λ(x : Bool) → λ(y : f x) → if x then
    "" else y

Error: <if> branches must have matching types
```

The `if/then/else` construction fails to typecheck even though we expect both `if` branches to return `Text` values. If we are in the `if/then` branch, we return a `Text` value (an empty string). If we are in the `if/else` branch, we return a value of type `f x`. That type depends on the value `x`. But we know that `x` equals `False` in that branch. So, the `else` branch should have the type `f False`, which is the same as the type `Text`. However, Dhall does not implement this logic and cannot see that both branches have the same type (`Text`).

Because of this and other limitations, Dhall can work productively with dependent types only in certain simple cases, such as validations of properties for function arguments.

Below in the chapter "Numerical algorithms" we will see an example of using dependent types for implementing a safe division operation.

2.4 The "assert" keyword and equality types

For values other than `Bool` and `Natural` numbers, equality testing is not available as a function. However, values of any type may be tested for equality at type-checking time via Dhall's `assert` feature. That feature is mainly intended for implementing sanity checks and unit tests:

```
let x : Text = "123"
let _ = assert : x ≡ "123"
in x ++ "1"
 -- This is a complete program that returns "1231".
```

The Dhall expression `a ≡ b` is a special *type* that depends on the values `a` and `b`. The type `a ≡ b` is different for each pair `a`, `b`. It is an example of a **dependent type**.

Types of the form `a ≡ b` are known as **equality types** because they have the following special properties:

The type `a ≡ b` has *no values* (is void) if `a` and `b` have different types, or the same types but different normal forms (as Dhall expressions). For example, the types `1 ≡ True` and `λ(x : Text) → λ(y : Text) → x ≡ λ(x : Text) → λ(y : Text) → y` are void. (We will never be able to create any values of those types.)

If `a` and `b` evaluate to the same normal form then the type `a ≡ b` is not void.

For example, the types `False ≡ False` and `123 ≡ 123` are not void.

Actually, types of the form `a ≡ a` are defined to be a unit type. That is, there always exists a single value of the type `a ≡ a`.

If we want to write that value explicitly, we use the `assert` keyword with the syntax `assert : a ≡ b`. This expression is valid only if the two sides are equal after reducing them to their normal forms. If the two sides are not equal after reduction to normal forms, the expression `assert : a ≡ b` will *fail to typecheck*, meaning that the entire program will fail to compile.

When an `assert` value is valid, we may assign that value to a variable:

```
let test1 = assert : 1 + 2 ≡ 0 + 3
```

In this example, the two sides of the type `1 + 2 ≡ 0 + 3` are equal after reducing them to normal forms. The resulting type `3 ≡ 3` is non-void and has a value. We assigned that value to `test1`.

It is not actually possible to print the value `test1` of type `3 ≡ 3` or to examine that value in any other way. That value *exists* (because the `assert` expression was accepted by Dhall), but that's all we know.

Some examples:

```
let x = 1
let y = 2
let _ = assert : x + 1 ≡ y      -- OK.
let print = λ(n : Natural) → λ(prefix : Text) → prefix ++ Natural/show n
let _ = assert : print (x + 1) ≡ print y    -- OK
```

In the last line, the `assert` expression was used to compare two partially evaluated functions, `print (x + 1)` and `print y`. The normal form of `print (x + 1)` is the Dhall expression `λ(prefix : Text) → prefix ++ "2"`. The normal form of `print y` is the same Dhall expression. So, the assertion is valid.

The fact that `assert` expressions are validated at typechecking time (before evaluating other expressions) has implications for using the `assert` feature in Dhall programs. For instance, one cannot use `assert` expressions for implementing a function for comparing two arbitrary values given as arguments.

To see why, try writing this code:

```
let compareTextValues : Text → Text → Bool
  = λ(a : Text) → λ(b : Text) →
    let _ = assert : a ≡ b    -- Type error: the two sides are not equal.
    in True
```

This code will fail to typecheck because, within the code of `compareTextValues`, the normal forms of the parameters `a` and `b` are just the *symbols* `a` and `b`, and those two symbols are not equal. Because this code fails to typecheck, we cannot use it to implement a function returning `False` when two text strings are not equal.

Another example: we cannot write a Dhall function that checks whether a string is empty, when that string is given as the function's parameter.

```
let isStringEmpty = λ(t : Text) →
  assert : t ≡ "" -- Type error: assertion failed.
```

This fails at typechecking time because the normal form of `t` is just the symbol `t` at that time, and that symbol is never equal to the empty string. An `assert` expression such as `assert : t ≡ ""` can be validated only when the value `t` is "statically known" in the scope of the expression. (We will discuss "statically known" values in more detail in the next subsection.)

These examples show that it rarely makes sense to use `assert` inside function bodies. The `assert` feature is intended for implementing unit tests or other static sanity checks. In those cases, we do not need to keep the value of each equality type. We just need to verify that all the equality types are not void. So, we will usually write unit tests like this:

```
let f = λ(a : Text) → "(" ++ a ++ ")" -- Define a function.
let _ = assert : f "x" ≡ "(x)"  -- OK.
let _ = assert : f "" ≡ "()"    -- OK.
-- Continue writing code.
```

2.5 Asserting that a value is statically known

We say that a value in a Dhall program is a **statically known value** if it is built up from literal constants, imported values, and/or other statically known values. To get some intuition, consider a typical Dhall program:

```
let Integer/add = https://prelude.dhall-lang.org/Integer/add
let Natural/equal = ...         -- Other library imports.

let k : Natural = ./size.dhall  -- Some value is imported.
let n : Natural = k + 123       -- Compute some more values.
let f = λ(x : Natural) →         -- Some custom function.
    let y = x * k
    in ...
let g = λ(a : Type) → ???        -- More custom functions.
let _ = assert : f n ≡ True     -- Perform some validations.
in ???                          -- Now compute the final result.
```

The values `f`, `g`, `k`, and `n` are statically known in this code. The value `k` is a direct import. The value`n` is computed from a literal number (`123`) and from `k`. The values `f` and `g` are literal function expressions defined at the top level. The bodies of the functions `f` and `g` may use `k`, `n`, and library imports. So, those functions are built up from statically known values.

Usually, values at the top level of a standalone Dhall program are statically known. Examples of values that are *not* statically known are the parameter `x` and the local variable `y` inside the body of the function `f` above. Generally, parameters of functions are not statically known within the scope of those functions.

The concept of "statically known values" is useful for understanding the usage of `assert` expressions. Consider an assertion of the form `assert : x ≡ y`. If `x` and `y` are statically known then the Dhall typechecker will validate that assertion by reducing both `x` and `y` to literal values. Typically, that's what the programmer expects when writing an `assert` expression. For this reason, `assert` expressions are most often used with statically known values.

Reducing an expression to a literal value at typechecking time is possible only if that expression is a statically known value.

When `x` or `y` is not statically known, the typechecker will only be able to reduce `x ≡ y` to a normal form containing symbolic variables. In most cases, this is not useful because it cannot be validated.

For example, here is an attempt to create a function `f` enforcing the condition that `f x` may be called only with nonzero `x`:

```
let f : Natural → Bool = λ(x : Natural) →
  let _ = assert : Natural/isZero x ≡ False
  in True -- Type error: assertion failed.
```

Within the body of this function, `x` is not a statically known value. The assertion of `Natural/isZero x ≡ False` will be examined at type-checking time, before the function `f` is ever applied to an argument. Dhall will try to validate the assertion of `Natural/isZero x ≡ False` by reducing both sides to the normal form. But the normal form of `Natural/isZero x` is just the symbolic expression `Natural/isZero x` itself. That symbolic expression is not equal to `False`, and so the assertion fails. We see that Dhall cannot validate this assertion in a meaningful way; this is almost surely not what the programmer expected.

A function may return the *type* `Natural/isZero x ≡ False` itself; that type is well-defined as a function of `x`. Then one can use that type to write assertions for statically known values:

```
let NonzeroNat : Natural → Type
  = λ(x : Natural) → Natural/isZero x ≡ False
let _ = assert : NonzeroNat 123   -- OK.
-- let _ = assert : NonzeroNat 0    -- This fails!
```

A curious consequence of the limitations of Dhall's evaluator and typechecker is that a function can require evidence that one of its parameters is a statically known value.

The implementation is based on the fact that Dhall cannot perform symbolic reasoning with `Natural` values other than in simplest cases. So, one can implement a constant function of type `Natural → Natural` that appears to perform nontrivial computations but actually always returns `0`. The function body can be chosen such that Dhall will be unable to detect statically that the function always returns `0`.

We can then apply that function to a `Natural`-valued parameter and assert that the result is indeed `0`. Dhall will accept the assertion only if the parameter can be reduced to a literal value. And that will be possible only when the parameter is statically known.

A suitable function is the following:

```
let zeroNatural : Natural → Natural = λ(x : Natural) → Natural/subtract 1
    (Natural/subtract x 1)
```

The function `zeroNatural` is a constant function that always returns `0`, but Dhall cannot recognize that property unless `zeroNatural` is applied to a statically known `Natural` argument:

```
let _ = assert : zeroNatural 0 ≡ 0
let _ = assert : zeroNatural 1 ≡ 0
let _ = assert : zeroNatural 2 ≡ 0
-- This fails with a type error:
-- λ(x : Natural) → assert : zeroNatural x ≡ 0
```

With this function, we can create a dependent type that always represents a valid assertion, but Dhall will be able to validate that assertion only for statically known values.

```
let StaticNatural : Natural → Type = λ(x : Natural) → zeroNatural x ≡ 0
let _ = assert : StaticNatural 123      -- OK.
-- This fails with a type error because ‘x‘ is not statically known:
-- λ(x : Natural) → assert : StaticNatural x
```

Now write a function taking a `Natural` argument `n` together with an evidence argument of type `StaticNatural n`:

```
let functionStaticOnly = λ(n : Natural) → λ(_ : StaticNatural n) → n + 1
```

We can call this function at the top level of a Dhall program, where all `Natural` values are statically known and an `assert : StaticNatural n` will always succeed.

```
let x = 123
let xIsStatic = assert : StaticNatural x
let y = functionStaticOnly x xIsStatic
let _ = assert : y ≡ 124
```

But it will be impossible to call `functionStaticOnly` within another function, without having an evidence value of type `StaticNatural x`:

```
-- Type error: assertion will fail.
let _ = λ(x : Natural) →
  let xIsStatic = assert : StaticNatural x  -- This fails.
  in functionStaticOnly x xIsStatic
```

A suitable equality type for `Integer` numbers is defined by:

```
let zeroInteger : Integer → Integer = λ(x : Integer) → Natural/toInteger
    (zeroNatural (Integer/clamp x))
let StaticInteger : Integer → Type = λ(x : Integer) → zeroInteger x ≡ +0
```

For `Bool` arguments, the code could look like this:

```
let zeroBool : Bool → Natural = λ(x : Bool) → zeroNatural (if x then 1 else 0)
let StaticBool : Bool → Type = λ(x : Bool) → zeroBool x ≡ 0
let _ = assert : StaticBool False  -- OK.
```

For `Text` arguments, we create a sequence of operations that will always return an empty string, and we assert on that property. But the Dhall interpreter is unable to validate that property symbolically. So, the assertion on `StaticText x` will fail unless `x` is a statically known value:

```
let emptyText : Text → Text = λ(x : Text) → Text/replace x "" (x ++ x)
let StaticText : Text → Type = λ(x : Text) → emptyText x ≡ ""
-- This fails with a type error because ‘x‘ is not statically known:
-- λ(x : Text) → assert : StaticText x
```

2.6 The universal type quantifier (∀) and the function symbol (λ)

Dhall uses the symbol `λ` (or equivalently the backslash `\`) to denote functions and the symbol `∀` (or equivalently the keyword `forall`) to denote *types* of functions.

- An expression of the form `λ(x : sometype1) → something2` is a function that can be applied to an argument in order to compute a result. Note that `something2` could be either a value or a type.

- An expression of the form `∀(x : sometype1) → sometype2` is always a *function type*: in other words, an expression that can be used as a type annotation for a function. In particular, `sometype2` must be a type.

Expressions of the form `∀(x : sometype1) → sometype2` may be used as type annotations for functions of the form `λ(x : sometype1) → something2`. (However, it is not important that the name `x` be the same in the function and in its type.)

For example, the function that appends `"..."` to a string argument is written like this:

```
let f = λ(x : Text) → x ++ "..."
```

The type of `f` can be written as `∀(x : Text) → Text` if we like. Let us write the definition of `f` together with a type annotation, to show how the type expression corresponds to the code expression:

```
let f : ∀(x : Text) → Text
  = λ(x : Text) → x ++ "..."
```

To summarize: `λ(x : a) → ...` is a function and can be applied to an argument. But `∀(x : a) → ...` is a type; it is not a function and cannot be applied to an argument.

The type expression `∀(x : Text) → Text` does not actually need the name `x` and can be also written in a shorter syntax as just `Text → Text`. But Dhall will internally rewrite that to the normal form `∀(_ : Text) → Text`.

A function type of the form `A → B` can be always rewritten in an equivalent but longer syntax as `∀(a : A) → B`. So, for instance, type expressions `∀(A : Type) → A → A` and `∀(A : Type) → ∀(x : A) → A` are equivalent. The additional name `x` could be chosen to help the programmer remember the intent behind that argument.

2.6.1 Example: Optional

To illustrate the difference between ∀ and λ compare the Dhall expressions `λ(r : Type) → Optional r` and `∀(r : Type) → Optional r`.

The expression `λ(r : Type) → Optional r` is equivalent to just the type constructor `Optional`. It is a function operating on types: it needs to be applied to a particular type in order to produce a result type. The expression `λ(r : Type) → Optional r` itself has type `Type → Type`.

The expression `∀(r : Type) → Optional r` has type `Type`. It is the type of functions having a type parameter `r` and returning a value of type `Optional r`. A value of type `∀(r : Type) → Optional r` must be a function written in the form `λ(r : Type) → ...` in some way. The code of such a function must work for all types `r` and must produce some value of type `Optional r`, no matter what the type `r` might be.

2.6.2 Example: polymorphic identity function

As another example to help remember the difference between ∀ and λ, look at the polymorphic identity function. That function takes a value `x` of an arbitrary type and again returns the same value `x`:

```
let identity
 : ∀(X : Type) → ∀(x : X) → X
  = λ(X : Type) → λ(x : X) → x
```

Here we denoted the type parameter by the capital `X`. (Dhall does not require that types be capitalized.)

Defined like this, `identity` is a function of type `∀(X : Type) → X → X`. The function itself is the expression `λ(X : Type) → λ(x : X) → x`.

The corresponding Haskell code is:

```
identity :: a -> a      -- Haskell.
identity = \x -> x
```

The corresponding Scala code is:

```
def identity[X]: X => X  = { x => x }     // Scala
```

In Dhall, the type parameters must be specified explicitly, both when defining a function and when calling it:

```
let identity = λ(X : Type) → λ(x : X) → x
let x = identity Natural 123  -- Writing just ‘identity 123‘ is a type error.
```

This makes Dhall code more verbose but also helps remove "magic" from the syntax, helping functional programmers to learn about more complicated types.

2.6.3 Example: "Invalid function output"

An expression of the form `λ(x : sometype1) → something2` is a function that can be applied to any `x` of type `sometype1` and will compute a result, `something2`. (That result could itself be a value or a type.) The *type* of the expression `λ(x : sometype1) → something2` is `∀(x : sometype1) → sometype2` where `sometype2` is the type of `something2`.

Another way to see that `∀` always denotes types is to write `∀(x : Text) → 123`. Dhall will reject that expression with the error message "Invalid function output". The expression `∀(x : Text) → something` is a *type* of a function, and `something` must be the output type of that function. So, `something` must be a type and cannot be a value. But in the example `∀(x : Text) → 123`, the output type of the function is specified as the number `123`, which is not a type.

In Dhall, this requirement is expressed by saying that `something` should have type `Type`, `Kind`, or `Sort`.

As another example of the error "Invalid function output", consider code like `∀(x : Type) → λ(y : Type) → x`. This code has the form `∀(x : Type) → something` where `something` is a function expression `λ(y : Type) → x`, which is not a type. So, a `λ` cannot be used in a curried argument after a `∀`.

Here are some valid examples using both `λ` and `∀` in curried arguments:

```
let _ = ∀(x : Type) → ∀(y : Type) → x
let _ = λ(x : Type) → ∀(y : Type) → x
let _ = λ(x : Type) → λ(y : Type) → x
```

2.7 Kinds and sorts

We have seen that in many cases Dhall treats types (such as `Natural` or `Text`) similarly to values. For instance, we could write `let N = Natural in ...` and then use `N` interchangeably with the built-in symbol `Natural`. The value `N` itself has a type that is denoted by the symbol `Type`. We may write it with a type annotation as `N : Type`.

```
⊢ :let N = Natural

N : Type
```

The symbol `Type` is itself treated as a special constant whose type is `Kind`:

```
⊢ :let p = Type

p : Kind
```

Other possible values of type `Kind` are "type constructor" kinds, such as `Type → Type`, as well as other type expressions involving the symbol `Type`.

```
⊢ :type (Type → Type) → Type

Kind
```

```
⊢ :type { a : Type }

Kind
```

As we have just seen, the type of `{ a = 1, b = Bool }` is the record type written in Dhall as `{ a : Natural, b : Type }`. The type of *that* is `Kind`:

```
⊢ :type { a : Natural, b : Type }

Kind
```

Any function that returns something containing `Type` will itself have the output type `Kind`:

```
⊢ :type λ(t : Bool) → if t then Type else Type → Type

∀(t : Bool) → Kind
```

Functions with parameters of type `Kind` can be used for creating complicated higher-order types. For example, here is a function that creates higher-order types of the form `k → k`, where `k` could be `Type`, `Type → Type`, or any other expression of type `Kind`:

```
⊢ :let f = λ(k : Kind) → k → k

f : ∀(k : Kind) → Kind

⊢ f Type

Type → Type

⊢ f (Type → Type → Type)

(Type → Type → Type) → Type → Type → Type
```

In turn, the symbol `Kind` is treated as a special value of type `Sort`. Other type expressions involving `Kind` are also of type `Sort`:

```
⊢ :type Kind

Sort

⊢ :type Kind → Kind → Type

Sort

⊢ :type λ(a : Kind) → a → a

∀(a : Kind) → Kind

⊢ :type ∀(a : Kind) → Kind
```

```
Sort
```

The symbol `Sort` is even more special: it *does not* itself have a type. Because of that, nearly any explicit usage of `Sort` will be a type error:

```
⊢ :let a = Sort

Error: <Sort> has no type, kind, or sort

⊢ λ(_ : Sort) → 0

Error: <Sort> has no type, kind, or sort
```

This feature of Dhall avoids the need for an infinite hierarchy of "**type universes**". That hierarchy is found in programming languages with full support for dependent types, such as Agda and Idris. In those languages, the type of `Type` is denoted by `Type 1`, the type of `Type 1` is `Type 2`, and so on to infinity. Dhall denotes `Type 1` by the symbol `Kind` and `Type 2` by the symbol `Sort`.

Dhall's type system has enough abstraction to support powerful types and to treat types and values in a uniform manner, while avoiding the complications with infinitely many type universes.

Because of this design, Dhall has very limited support for working with the symbol `Kind` itself. Little can be done with Dhall expressions such as `Kind` or `Kind → Kind`. One can assign such expressions to variables, one can use them for type annotations, and that's about it.

For instance, it is a type error to write a function that returns the symbol `Kind` as its output value:

```
⊢ :let a = Kind

a : Sort

⊢ :let f = λ(_: Bool) → a

Error: <Sort> has no type, kind, or sort
```

This error occurs because Dhall requires a function's type *itself* to have a type. The symbol `Kind` has type `Sort`, so the type of the function `f = λ(_: Bool) → Kind` is `Bool → Sort`. But the symbol `Sort` does not have a type, and neither does the expression `Bool → Sort`. Dhall raises a type error because the function `f`'s type (which is `Bool → Sort`) does not itself have a type.

For the same reason, Dhall will not accept the following function parameterized by a `Kind` value:

```
⊢ :let f = λ(k : Kind) → ∀(b : Kind) → k → b

Error: <Sort> has no type, kind, or sort
```

This prevents Dhall from defining recursive kind-polymorphic type constructors.

For example, Dhall cannot implement an analog of `List` that works with types of arbitrary kinds.

There was at one time an effort to change Dhall and to make `Kind` values more similar to `Type` values, so that one could have more freedom with functions with `Kind` parameters. But that effort was abandoned after it was discovered that it would break the consistency of Dhall's type system[1].

Nevertheless, it is perfectly possible to define a list of types of a specific kind. For instance, one can define a list of types of kind `Type`, such as `Natural`, `Bool`, and `Text`. Separately, one can define a list of types of kind `Type → Type`, such as `Optional` and `List`. Those definitions will use similar code, but it will not be possible to refactor them into special cases of a more general kind-polymorphic code. (This limitation does not appear to be particularly important for the practical use cases of Dhall.)

[1]https://github.com/dhall-lang/dhall-haskell/pull/563#issuecomment-426474106