Monad (functional programming)

From Wikipedia, the free encyclopedia

In functional programming, a *monad* is a structure that represents computations defined as sequences of steps: a type with a monad structure defines what it means to chain operations, or nest functions of that type together. This allows the programmer to build pipelines that process data in steps, in which each action is decorated with additional processing rules provided by the monad.[1] As such, monads have been described as "programmable semicolons"; a semicolon is the operator used to chain together individual statements in many imperative programming languages,[1] thus the expression implies that extra code will be executed between the statements in the pipeline. Monads have also been explained with a physical metaphor as assembly lines, where a conveyor belt transports data between functional units that transform it one step at a time.[2] They can also be seen as a functional design pattern to build generic types.[3]

Purely functional programs can use monads to structure procedures that include sequenced operations like those found in structured programming.[4][5] Many common programming concepts can be described in terms of a monad structure, including side effects such as input/output, variable assignment, exception handling, parsing, nondeterminism, concurrency, and continuations. This allows these concepts to be defined in a purely functional manner, without major extensions to the language's semantics. Languages like Haskell provide monads in the standard core, allowing programmers to reuse large parts of their formal definition and apply in many different libraries the same interfaces for combining functions.[6]

Formally, a monad consists of a type constructor `M` and two operations, bind and return (where return is often also called unit):

The *return* operation takes a value from a plain type and puts it into a monadic container using the constructor, creating a monadic value.
The *bind* operation performs the reverse process, extracting the original value from the container and passing it to the associated next function in the pipeline, possibly with additional checks and transformations.

The operations must fulfill several properties to allow the correct composition of monadic functions (i.e. functions that use values from the monad as their arguments or return value). Because a monad can insert additional operations around a program's domain logic, monads can be considered a sort of aspect-oriented programming.[7] The domain logic can be defined by the application programmer in the pipeline, while required aside bookkeeping operations can be handled by a pre-defined monad built in advance.The name and concept comes from the eponymous concept (monad) in category theory, where monads are one particular kind of functor, a mapping between categories; although the term monad in functional programming contexts is usually used with a meaning corresponding to that of the term strong monad in category theory.